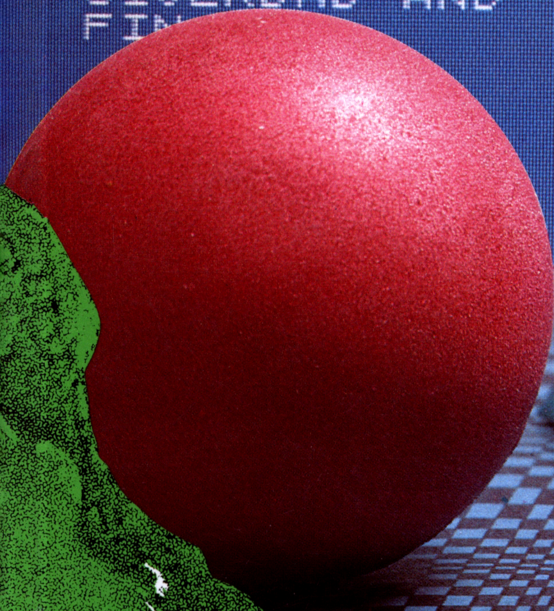
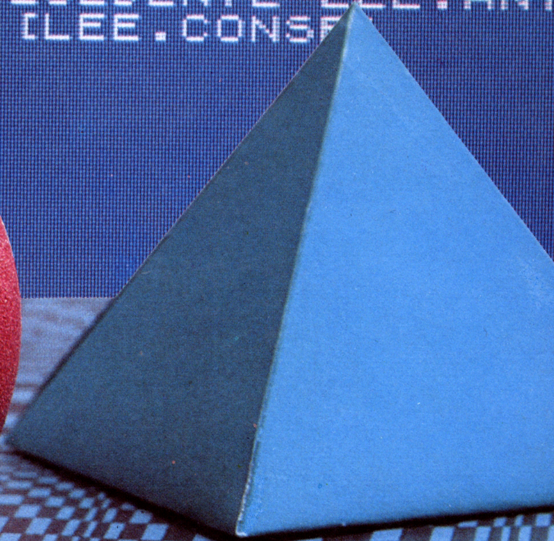
Joaquín D'Opazo Alvarez/Grupo Golem

Programación en LOGO







PROGRAMACION EN LOGO

Con la colaboración de:

Eliseo BORRAS VESES
Francisco BURGUET MORET
Vicente GISBERT GINER
Rafael IBORRA SERRANO
Magdalena MORATA CUBELLS
María Teresa NIETO APARICI
Agustín SANCHEZ MAMBRILLA
Mercedes TESTAL PARRILLA

todos ellos miembros del Grupo GOLEM.

Programación en LOGO

Joaquín D'Opazo Alvarez

Grupo GOLEM



MICROINFORMATICA

Diseño de colección: Antonio Lax
Diseño de cubierta: Narcís Fernández

Primera reimpresión, octubre 1986

Villafranca, 22. 28028 Madrid
Depósito legal: M. 32.831-1986
ISBN: 84-7614-033-9
Printed in Spain
Imprime: Anzos, S. A. - Fuenlabrada (Madrid)

Indice

Introducción

Más que un lenguaje de programación, LOGO es una concepción de la enseñanza, basada en la utilización del ordenador como herramienta de trabajo, como un amplificador de la inteligencia, que permita a las personas explorar, investigar, resolver problemas; es decir, aprender.

Los diseñadores de LOGO pensaron en él como una herramienta educativa "sin límite inferior ni superior", y, en efecto, desde los dibujos de alumnos de Preescolar hasta la Teoría de la Relatividad Generalizada, desde los programas elementales de listas hasta los sistemas expertos (programas que hacen deducciones lógicas), todo tiene cabida en LOGO.

Hay tres características de LOGO que es interesante señalar. En primer lugar, se trata de un lenguaje "interactivo", es decir, cualquiera de las órdenes de que dispone el lenguaje (las llamadas primitivas), o de las que se pueden crear utilizando éstas, puede ejecutarse sin más que escribirla en el teclado. En segundo lugar, se trata de un "lenguaje de procedimientos", esto es, un programa LOGO se construye combinando primitivas en grupos llamados "procedimientos", que a su vez pueden agruparse formando otros procedimientos, y así sucesivamente hasta alcanzar cualquier grado de complejidad; el editor de LOGO simplifica enormemente esta tarea. La tercera característica es que dispone del "micromundo" de la "tortuga", pequeño habitante de la pantalla del ordenador que obedece fielmente las órdenes que se le

dan, y cuya popularidad ha alcanzado semejantes cotas que otros lenguajes han incorporado parte de estas habilidades gráficas a sus propias estructuras. Precisamente vamos a utilizar este simpático animalito para empezar a trabajar con LOGO.

La ventaja de los "lenguajes de alto nivel", entre los que se incluye LOGO, es que permiten escribir los programas de un modo muy similar a como habla (piensa) el programador. Por ello es conveniente que las "palabras" del lenguaje pertenezcan a la lengua materna del usuario. En este manual vamos a emplear una versión castellana de LOGO implementada en los ordenadores APPLE-II e IBM-PC por el Grupo Golem de Valencia.

ANTE
TE
ANTECEDEN
LEELISTA
TE = CD
CHRΣ"ANTE
:ANTECED
AND ILEE

1 Los primeros pasos

Puesto en marcha el sistema (consulta el manual de tu máquina), y tras el mensaje de bienvenida a LOGO, éste te "invita" a darle órdenes, haciendo aparecer el símbolo "?", con el cursor a su lado. Ya puedes empezar a escribir. Prueba con

ESCRIBE

```
ESCRIBE "HOLA
```

y pulsa la tecla "retorno", ↵, que significa "fin de línea" (equivalente al retorno de carro de una máquina de escribir).

LOGO obedecerá tu orden escribiendo en la pantalla

```
HOLA
?
```

y el signo "?" volverá a aparecer en la línea siguiente, invitándote a continuar. Prueba con

```
ESCRIBE [ESTO ES SOLO EL PRINCIPIO]
```

y pulsa "retorno". Verás aparecer en la pantalla

```
ESTO ES SOLO EL PRINCIPIO
?
```

y otra vez el signo "?". Hagamos un poco de aritmética. Prueba con

```
ESCRIBE 17*18
```

y pulsa "retorno". Aparecerá

```
306
?
```

Si quieres hacer algo más de aritmética, continúa con ella; en caso contrario, vamos a presentarte la tortuga.

Gráficos

Escribe

LG

```
LG
```

abreviatura de "limpia gráficos", y pulsa "retorno". Antes de continuar, como esto del retorno se va haciendo muy pesado, no vamos a repetirlo más. Ya lo sabes: después de escribir cada orden, pulsa esta tecla, ya que mientras no lo hagas LOGO permanecerá a la espera. Continuemos. Como observarás, la pantalla ha cambiado radicalmente. El pequeño triángulo que aparece en el centro, apuntando a la parte superior de la pantalla, es la famosa tortuga. Observa también que el signo ? aparece ahora en la parte inferior, lo cual indica que espera tus órdenes. Este tipo de pantalla, en la que coexisten gráficos y texto, es la llamada MIXTA. Escribe

GRAFICOS

```
GRAFICOS
```

y observa que el cursor desaparece; toda la pantalla está dedicada a gráficos. ¿Qué pasa ahora con las órdenes y los mensajes de LOGO? No te preocupes. Escribe con mucho cuidado, ya que no verás las letras saliendo en la pantalla, la palabra MIXTA, y todo volverá a ser como antes; además, sí verás esta palabra. Si escribes

TEXTOS

```
TEXTOS
```

advertirás cómo la tortuga desaparece, quedando la pantalla reservada a los textos. Ya conoces los tres tipos de presentación en pantalla de LOGO y cómo pasar de uno a otro; así pues, sitúate en la pantalla mixta porque vamos a empezar a mover la tortuga. Para ello puedes

utilizar cuatro instrucciones: AVANZA, RETROCEDE, IZQUIERDA y DERECHA. Escribe

AVANZA
RETROCEDE
IZQUIERDA
DERECHA

```
AVANZA
```

y te encontrarás con un mensaje de LOGO:

```
FALTAN DATOS PARA AVANZA
```

Normal; hay que decirle cuánto tiene que avanzar. Prueba sucesivamente las instrucciones

```
IZQUIERDA 45
AVANZA 50
DERECHA 45
AVANZA 30
DERECHA 90
```

Observarás que todas necesitan un número, un "dato de entrada". En AVANZA y RETROCEDE este número es una longitud, medida en "pasos" de tortuga, mientras que en IZQUIERDA y DERECHA es un ángulo, medido en grados. ¿Qué instrucciones tendrías que dar a la tortuga para que volviese a su posición inicial? Piénsalo un poco antes de seguir leyendo. Por cierto, ¿no te has equivocado al escribir alguna instrucción? Si ha sido así, no te preocupes; en los manuales de las máquinas tienes explicaciones sobre la forma de corregir.

Volviendo a las instrucciones que se te pedían, hay un modo muy sencillo de resolver el problema, ya que basta con escribir:

CENTRO

```
CENTRO
```

Haz unos cuantos dibujos, los que te apetezcan, utilizando las primitivas que conoces, sin olvidar LG (recuerda: limpia gráficos). Por cierto, ¿qué longitud y qué anchura tiene tu pantalla, medida en pasos de tortuga? Hay otras primitivas que pueden resultarte útiles para dibujar. Escribe

SL

```
AVANZA 50 IZQUIERDA 60
SL
AVANZA 50
```

¡Sorpresa! SL significa "sin lápiz".

CL

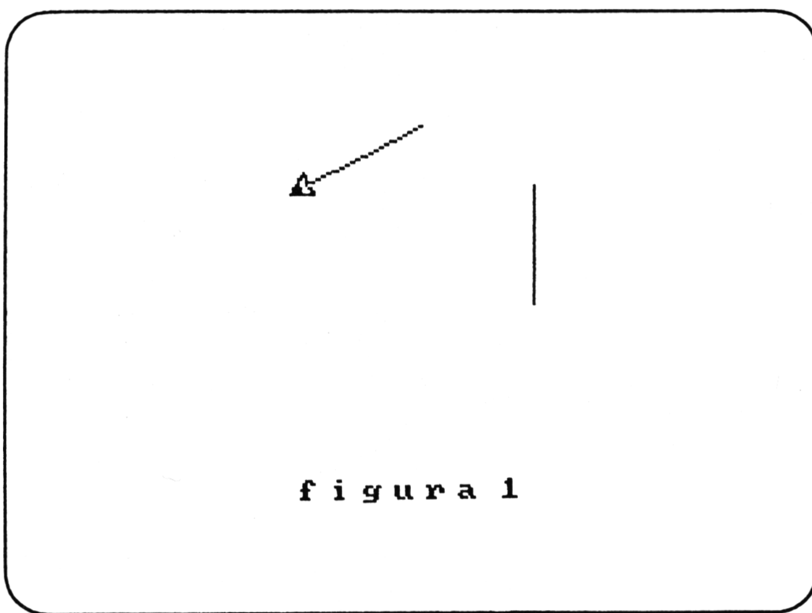
```
IZQUIERDA 60 CL
```

¿Qué significará CL?

```
AVANZA 50
```





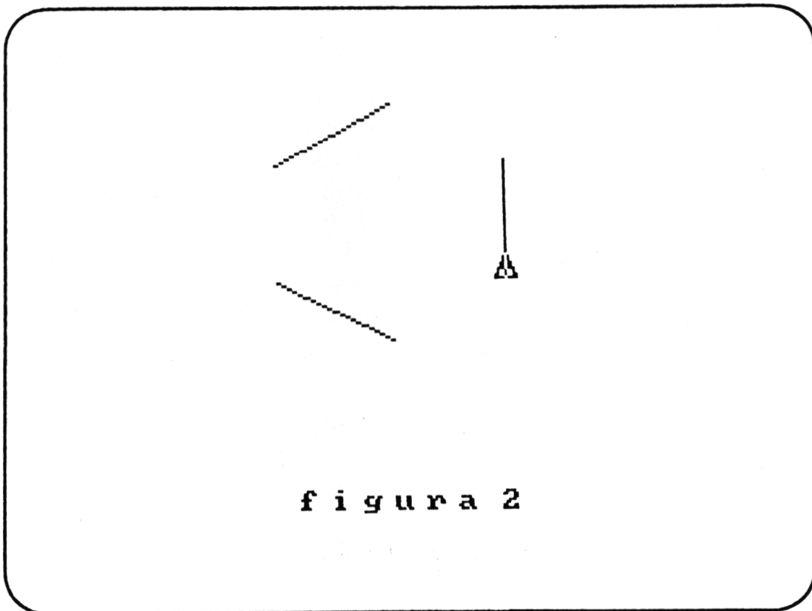



Ya lo sabes.

Continúa escribiendo las instrucciones oportunas para completar el hexágono.





Escribe ahora

LIMPIA

```
LIMPIA
```

y observarás la diferencia existente entre esta primitiva y LG.

Una forma de repetir: iteración

Limpia la pantalla y escribe, en la misma línea, LG y, a continuación, la pareja de instrucciones AVANZA 50 DERECHA 60, repetida seis veces. Está claro que todas estas instrucciones no caben en una sola línea de la pantalla, pero haciéndolo observarás cómo LOGO hace aparecer una marca cuando llegas al final de la línea para indicarte que "la cosa continúa" (la marca es un signo de admiración, "!", en el APPLE, y una flecha hacia la derecha, en el IBM). Dicho sea de paso, ya has visto que pueden escribirse varias instrucciones en la misma línea, y que son ejecutadas en el orden en que han sido escritas.

Otra posibilidad interesante es la siguiente; escribe

```
AVANZA 50 DERECHA 90
```

y pulsa "retorno". Si ahora presionas, en el APPLE, la tecla marcada con "Ctrl" y, manteniéndola apretada, pulsas la tecla de la letra Y, observarás que vuelve a aparecer de nuevo la línea de instrucciones que escribiste; para ejecutarla basta con que pulses "retorno". Vuelve a utilizar Ctrl-Y para cerrar la figura. En el IBM es más fácil: basta, para obtener la repetición, con presionar la tecla marcada con "F3". El caso es que, si deseas repetir un par de instrucciones como ésta, puedes encargar toda la tarea a LOGO escribiendo

REPITE

```
REPITE 4 [AVANZA 50 DERECHA 90]
```
(figura 3)

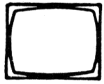
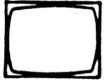
Por supuesto, puedes reiterar repeticiones; prueba con

```
REPITE 8 [REPITE 4 [AVANZA 50 DERECHA 90] DERECHA 45]
```
(figura 4)

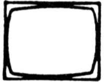
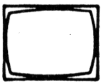
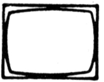
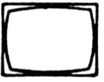
Prueba con otros valores y, si "no te quedan bien", observa el número de repeticiones y el ángulo que se gira después de dibujar el cuadrado.

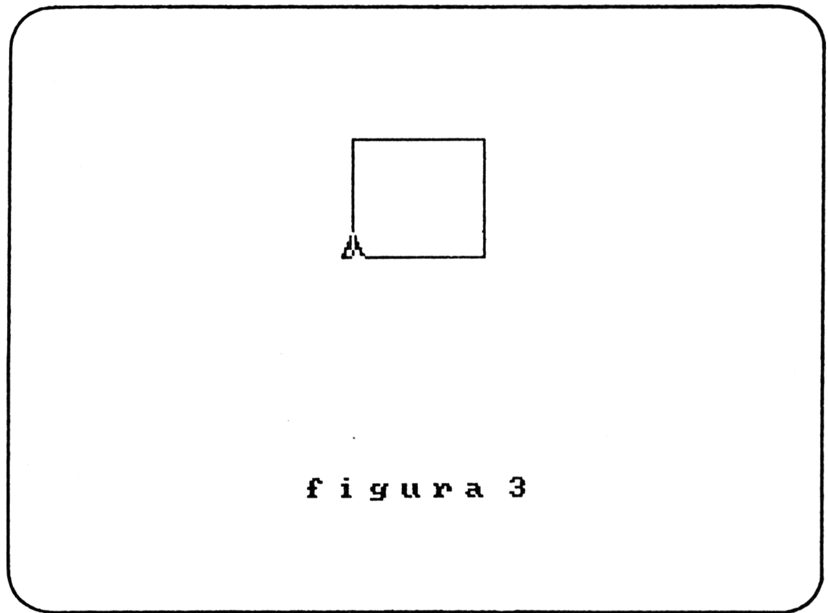
Intenta esta otra:

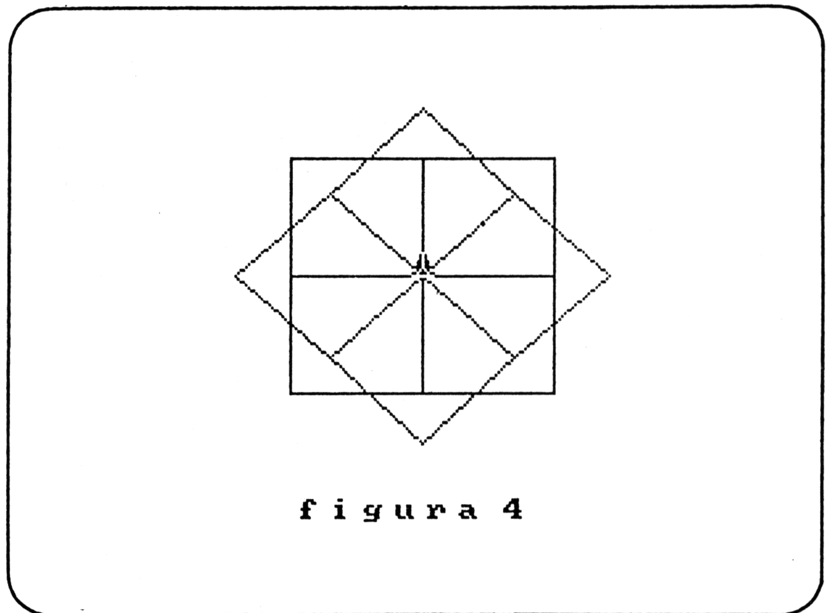
```
REPITE 18 [AVANZA 60 DERECHA 20]
```
(figura 5)



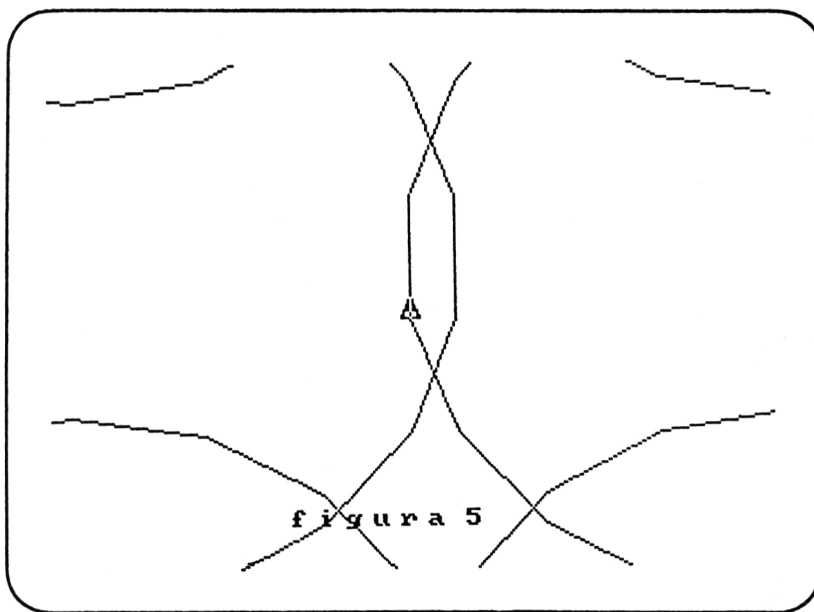




BARRERA

Ahora ya sabes lo que le pasa a la tortuga cuando desaparece por un lado de la pantalla: aparece por el otro. Si no quieres que ocurra esto, escribe BARRERA y, a continuación (usando Ctrl-Y o F3), la repetición anterior. Verás que la tortuga está confinada a los límites





de la pantalla y cualquier intento de salir de ellos produce un mensaje de error:

```
TORTUGA FUERA DE LIMITES
```

Aún hay otra forma de proceder; escribe

VENTANA

```
VENTANA REPITE 18 [AVANZA 60 DERECHA 20]
```
(figura 6)

Como ves, la pantalla se ha transformado ahora en una ventana por donde se ve parte del "mundo" de la tortura. Si quieres regresar a la situación inicial, que es "el valor por defecto" de la pantalla, basta con que escribas

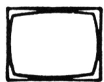
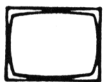
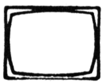
VUELVE

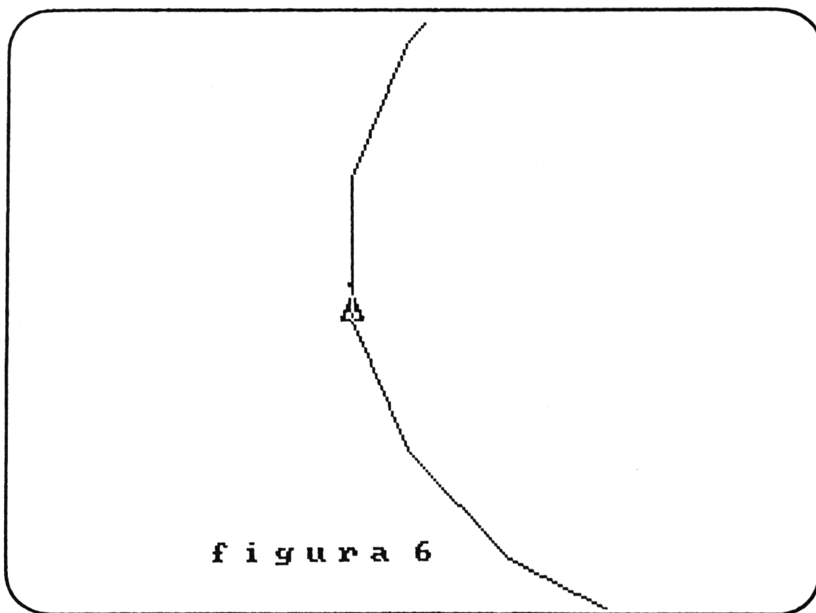
```
VUELVE.
```

Color

Si dispones de un monitor en color, puedes utilizar el color del lápiz de la tortuga y el del fondo de la pantalla para conseguir efectos espectaculares. Las primitivas VALCOLOR y VALFONDO no tienen





entradas, pero sí salidas: los números correspondientes a los colores. Si, tras poner en marcha el sistema, escribes

VALCOLOR
VALFONDO

```
ESCRIBE VALCOLOR ESCRIBE VALFONDO
```

obtendrás como respuesta la pareja de números 3 y 0 (si quieres saber exactamente cuáles son los colores de que se dispone en cada máquina, consulta el manual de la misma). Las primitivas COLOR y FONDO te permiten cambiar los colores del lápiz de la tortuga y del fondo de la pantalla. Incluso en monitores de blanco y negro hay un efecto interesante de la primitiva COLOR: si dibujas con el mismo color que el del fondo, entonces no se ve nada. Prueba lo siguiente

COLOR
FONDO

```
REPITE 2 [REPITE 4 [AVANZA 50 DERECHA 90] COLOR 0] COLOR 3
```

Como ves, puedes borrar un dibujo volviendo a dibujar sobre él con el mismo color que el del fondo. El mismo efecto se consigue utilizando la primitiva BL, “borra lápiz”; prueba

BL

```
REPITE 2 [REPITE 4 [AVANZA 50 DERECHA 90] BL] CL
```

Quizá te haya extrañado el aspecto “rectangular” que tienen los cuadrados en la pantalla del APPLE; en efecto, los pasos de la tortuga son más cortos hacia el norte o sur (de la pantalla, claro) que hacia el este o el oeste. Si le preguntas a LOGO el valor de la escala:

VALESCALA

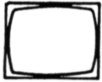
```
ESCRIBE VALESCALA
```

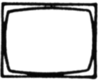
te contestará con el número 0.8. Ello quiere decir que diez pasos en la dirección norte-sur equivalen a ocho en la este-oeste.
Puedes modificar tal estado de cosas con la orden:

ESCALA

```
ESCALA 1
```

Realiza algunas pruebas más con las primitivas que ya conoces. ¿Podrías dibujar algo así?



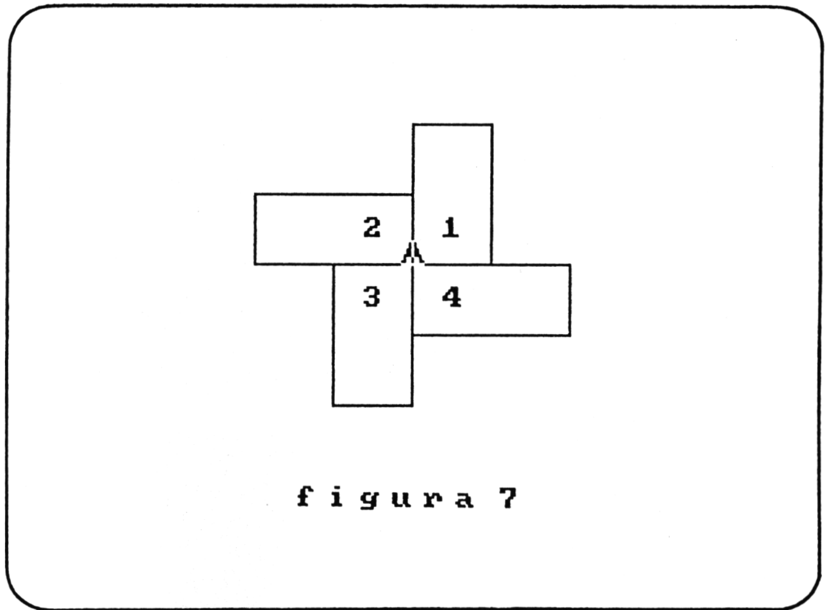



figura 7

Los números indican simplemente el orden en que se han dibujado los rectángulos.

Vamos a examinar un poco el asunto. Dibujado el primer rectángulo, bastaría la instrucción

```
IZQUIERDA 90
```

seguida de las de dibujo de un nuevo rectángulo, para tener el 1 y el 2 hechos; otra

```
IZQUIERDA 90
```

y nuevas instrucciones de rectángulo producirían el 3, etc.

Supón por un momento que al dar la orden

```
RECTANGULO
```

LOGO te dibujase uno. En ese caso, para trazar la figura bastaría con escribir

```
REPITE 4 [RECTANGULO IZQUIERDA 90]
```

Desde luego no existe la primitiva RECTANGULO, pero eso no es ningún problema: se fabrica y arreglado; es decir, se le enseña a LOGO lo que es un rectángulo. De ese modo RECTANGULO será un procedimiento que LOGO ejecutará como si fuese una primitiva. Vamos a ver entonces cómo se construyen los procedimientos, lo cual merece capítulo aparte.

ANTE
NTE
LANTECEDEN
E LEELISTA
NTE = []
CHAZ"ANTE
E :ANTECED
D AND ILEE

2 Programando con procedimientos

Para definir un procedimiento lo mejor es utilizar el editor del que dispone LOGO. Para ponerlo en marcha se utiliza la primitiva EDITA; escribe

EDITA

```
EDITA "RECTANGULO
```

Como verás, el cambio en la pantalla es radical; la línea en campo negativo te indica que estás en el editor, y no desaparecerá mientras estés en él. La primera línea es una atención suya: es la línea del título. La palabra SEA es una primitiva que sirve para definir procedimientos, y hará que RECTANGULO sea lo que vas a escribir a continuación. Teclea tú las instrucciones necesarias y termina con la palabra FIN. Si piensas en los movimientos que ejecutaría una tortuga para dibujar un rectángulo escribirás algo parecido a esto (después de pulsar "retorno" al final de cada línea):

SEA

```
SEA RECTANGULO
REPITE 2 [AVANZA 60 DERECHA 90 AVANZA 3!
0 DERECHA 90]
FIN
```

La admiración que observas la pone el editor para indicar que la línea de instrucciones continúa en la siguiente línea de pantalla, ya que en principio sólo caben en ésta cuarenta caracteres; esto en

APPLE, ya que IBM utiliza una pequeña flecha hacia la derecha. De todos modos, es la última vez que la ponemos; en lo sucesivo escribiremos las líneas de instrucciones tal como las necesitemos. Cuando las teclees en tu máquina verás aparecer el símbolo, así que no te sorprendas.

Para que el procedimiento que acabas de escribir pase a figurar entre los ejecutables por la máquina basta con pulsar:

CTRL-C, en el APPLE
ESC, en el IBM

con lo cual desaparece la pantalla del editor y aparece el mensaje

```
RECTANGULO DEFINIDO
```

Si ahora quieres dibujarlo no tienes más que dar la orden:

```
RECTANGULO
```

Para dibujar la figura que habíamos indicado es suficiente escribir

```
REPITE 4 [RECTANGULO IZQUIERDA 90]
```

Te proponemos que definas un procedimiento (puedes llamarlo MOLINILLO1) que se encargue de dibujar la figura al dar la orden

```
MOLINILLO1
```

En lugar de repetir cuatro veces un giro de 90 grados, podrías repetir ocho veces un giro de 45 grados. Para hacerlo edita MOLINILLO1 y cámbialo hasta que obtengas algo semejante a:

```
SEA MOLINILLO2
REPITE 8 [RECTANGULO IZQUIERDA 45]
FIN
```

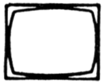
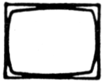
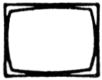
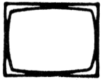
y sal del editor, con lo cual aparecerá el mensaje:

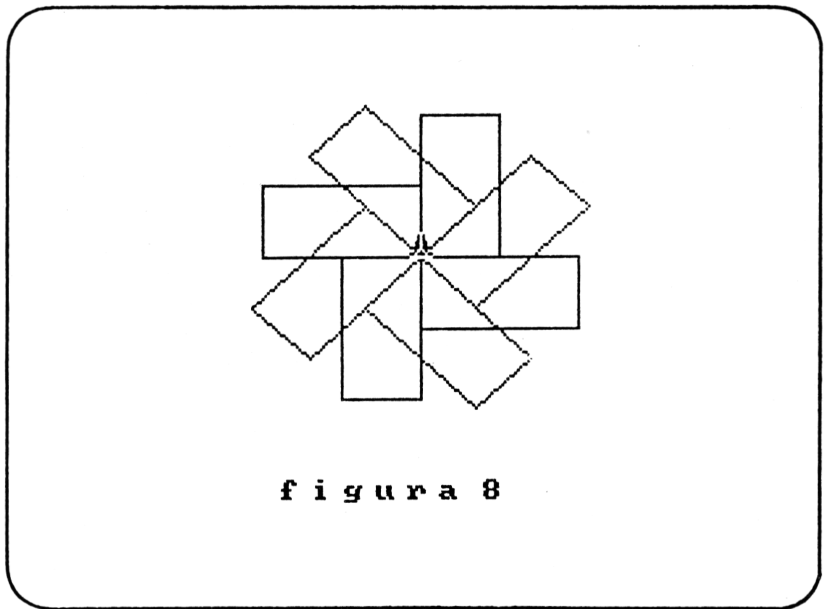
```
MOLINILLO2 DEFINIDO
```
(figura 8)

Ejecútalo para que compruebes lo que hace. Por supuesto, LOGO sabe ahora lo que son MOLINILLO1 y MOLINILLO2.

ECTS

Cada vez que quieras ver cuántos procedimientos "se sabe" LOGO, usa la primitiva ECTS (escribe títulos) y te contestará con los nombres que, en ese momento, conoce. Si deseas ver el texto de un

ECP

procedimiento puedes usar EDITA o ECP (escribe procedimiento) en la forma

```
ECP "MOLINILLO1
```

o bien,

```
ECP [MOLINILLO1 MOLINILLO2]
```

Prueba ambas para ver qué sucede.

Datos variables

Ya has dibujado un cuadrado. Atrévete con un pentágono, un hexágono, un heptágono y un octógono. Estos procedimientos tienen un inconveniente: siempre hacen lo mismo. Sería interesante poder variar a voluntad el tamaño del lado o el número de lados de un polígono. Has visto que algunas primitivas de la tortuga admiten distintas entradas; esto sería cómodo en los procedimientos, ya que, variando estos datos, podrían conseguirse distintos efectos. Pasa al editor y modifica RECTANGULO para que quede así:

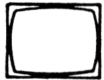
```
SEA RECTANGULO :LADO1 :LADO2
REPITE 2 [AVANZA :LADO1  DERECHA 90  AVANZA  :LADO2  DERECHA
90]
FIN
```

La forma en que se ha definido RECTANGULO es la habitual para procedimientos con datos. No tienes más que elegir los nombres que vas a darles y escribirlos en la línea del título precedidos por el símbolo ":" (en lugar de leer "dos puntos lado1, dos puntos lado2", acostúmbrate a leer "valor de lado1, valor de lado2").

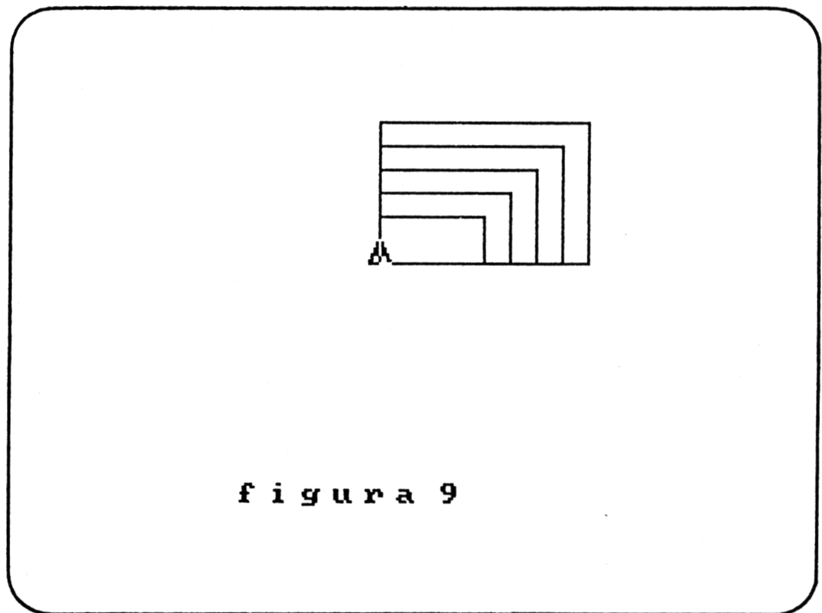
Si, una vez que hayas salido del editor, te limitas a escribir RECTANGULO, te encontrarás con el mensaje FALTAN DATOS PARA RECTANGULO, lo mismo que si escribes sólo AVANZA.

Prueba, por ejemplo, sin borrar la pantalla gráfica, lo siguiente:

```
RECTANGULO 20 40
RECTANGULO 30 50
RECTANGULO 40 60
RECTANGULO 50 70
RECTANGULO 60 80
```







figura 9

Hay algo que conviene dejar claro sobre los nombres de los datos en los procedimientos: aunque los nombres de éstos quedan incorporados al vocabulario de LOGO, los de los datos (también llamados parámetros) son "almacenes privados" (variables locales) del procedi-

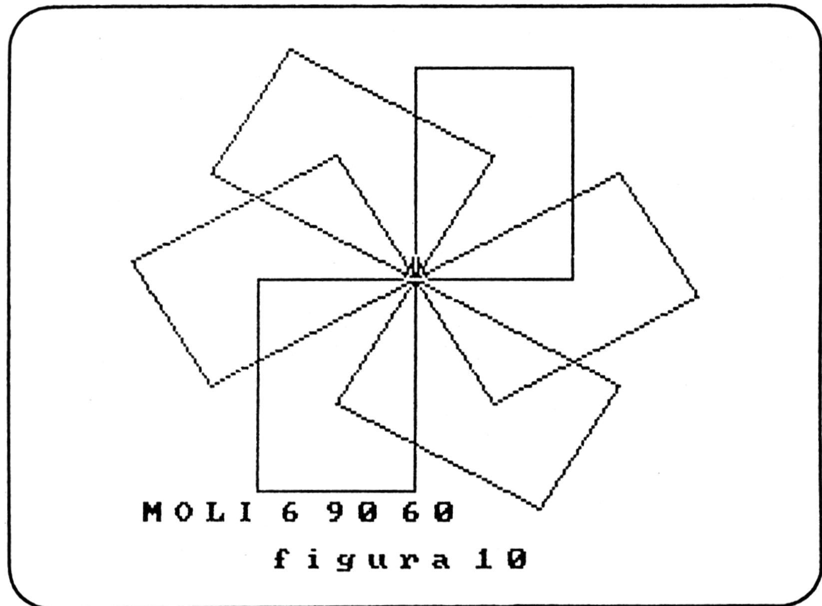
miento que los usa, de modo que puedes utilizar los mismos nombres en diferentes procedimientos, sin que interfieran unos con otros.

Puedes considerar un procedimiento como una "caja negra" que dibuja un rectángulo cuando se le dan los lados, sin tener que preocuparse de más. Esta es una idea crucial en la programación: cada vez que se define un procedimiento se le puede usar como bloque para la construcción de otros más complejos.

Siguiendo con el ejemplo que ya hemos visto, puede definirse el procedimiento MOLINILLO usando el nuevo RECTANGULO. Aprovechando la posibilidad de utilizar parámetros puedes indicar el número de rectángulos a dibujar, al que llamaremos NUMERO. De RECTANGULO basta con saber que tiene dos entradas —las longitudes de los lados— y que dibuja un rectángulo, dejando la tortuga donde estaba al comienzo del dibujo. Aquí tienes una posibilidad:

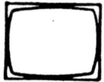
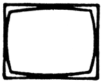
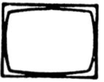
```
SEA MOLINILLO :NUMERO :LADO1 :LADO2
REPITE   :NUMERO   [RECTANGULO   :LADO1   :LADO2   IZQUIERDA
360/:NUMERO]
FIN
```





Cuando termina el dibujo de MOLINILLO, la tortuga queda en idéntica posición que estaba al empezar; por tanto, disponiendo de este procedimiento puedes dibujar varios molinillos en la pantalla. Por ejemplo:

```
SEA MOLINILLOS :NUMERO :LADO1 :LADO2
SL IZQUIERDA 90 AVANZA 100 DERECHA 90 CL
MOLINILLO :NUMERO :LADO1 :LADO2
SL CENTRO CL
MOLINILLO :NUMERO :LADO1 :LADO2
SL DERECHA 90 AVANZA 100 IZQUIERDA 90 CL
MOLINILLO :NUMERO :LADO1 :LADO2
FIN
```

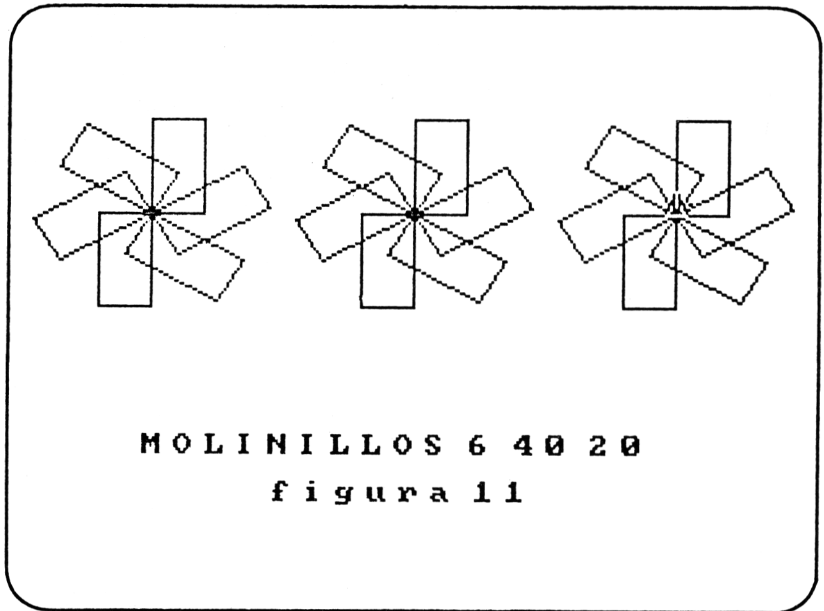
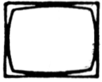
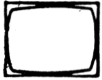
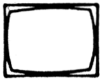



figura 11

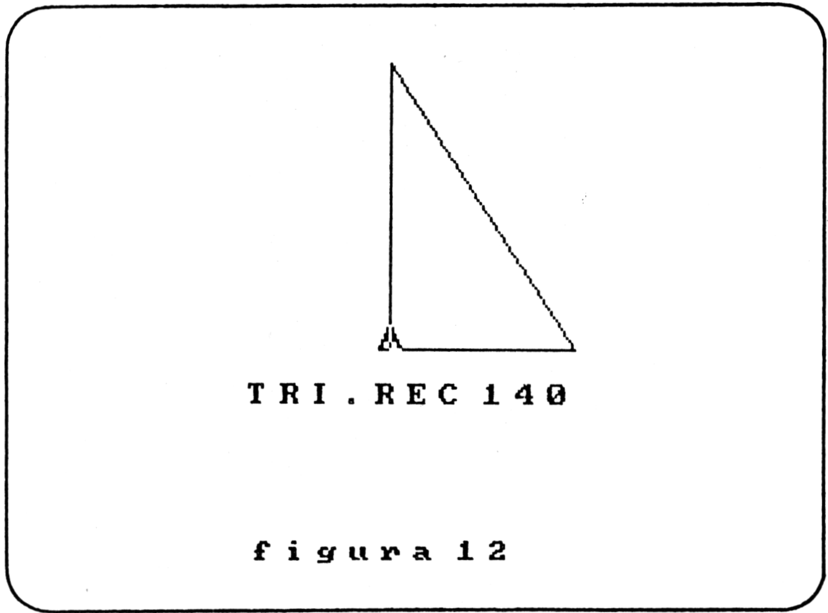
Intentaremos ahora dibujar un triángulo rectángulo con un ángulo de 30 grados. Daremos como dato la hipotenusa, representada por el parámetro HIPO. Recuerda cómo se calculan los catetos a partir de la hipotenusa, y utiliza la primitiva RAC que devuelve la raíz cuadrada de un número, así como las primitivas producto * y cociente /. Nos quedaría algo como:

```
SEA TRI.REC :HIPO
AVANZA (:HIPO * RAC 3) / 2
DERECHA 150
AVANZA :HIPO
DERECHA 120
AVANZA :HIPO / 2
DERECHA 90
FIN
```

Analiza este procedimiento viendo cómo se construye la figura. Para ello conviene imaginar el movimiento de la tortuga al dibujar el triángulo; observa que el ángulo que describe la tortuga no coincide con el ángulo interior que forman los lados del triángulo.





Puedes definir, usando TRI.REC, un procedimiento que dibuje molinillos de aspas triangulares.

Otra forma de repetir: recursión

Ya has utilizado la primitiva REPITE en varios procedimientos. Si has dibujado los polígonos que te indicamos anteriormente, te percatarás ahora de que la utilización de parámetros variables permite usar un solo procedimiento para la construcción de todos ellos. Podrías empezar por

```
SEA POLIGONO :NUMLADOS :LADO :ANGULO
REPITE :NUMLADOS [AVANZA :LADO DERECHA :ANGULO]
FIN
```

Si lo piensas un poco, observarás que tres variables son demasiadas, ya que existe una relación entre el ángulo y el número de lados, por lo que NUMLADOS es innecesaria; por tanto, podrías dejarlo en

```
SEA POLIGONO :LADO :ANGULO
REPITE <número de lados> [AVANZA :LADO DERECHA :ANGULO]
FIN
```

Lo de ⟨número de lados⟩ puedes escribirlo tranquilamente en tu libreta cuando estés diseñando un procedimiento. Significa que aplazas el momento en que escribirás algo que LOGO pueda entender. Cuando hayas descubierto la relación podrás editar y ejecutar el procedimiento, pero eso no ocurrirá hasta que pienses un poco y después de realizar algunas pruebas.

Observa que REPITE es útil para este propósito, siempre que conozcas cuántas veces tienes que repetir; pero ¿y si no lo sabes? Hay una forma cómoda de repetición que puede utilizar LOGO; prueba este procedimiento

```
SEA POLIGONO :LADO :ANGULO
AVANZA :LADO
DERECHA :ANGULO
POLIGONO :LADO :ANGULO
FIN
```







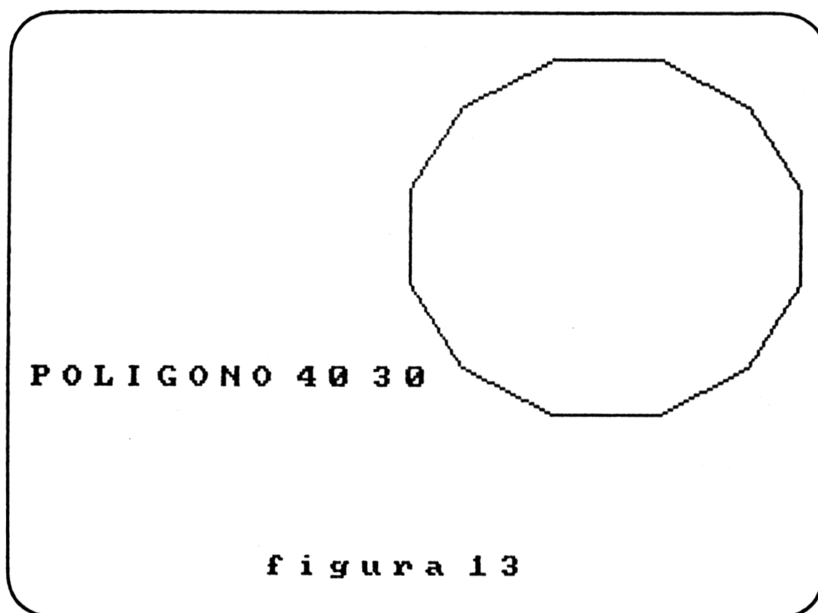


figura 13

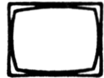
Cuando te hayas cansado de ver cómo la tortuga da vueltas y más vueltas, usa CTRL-G (en el APPLE) o CTRL-BREAK (en el IBM) para detenerla. Utiliza distintos valores de ángulo, con intención de resolver el problema que se planteó antes. Ahora que has observado lo que hace, entenderás mejor cómo funciona.

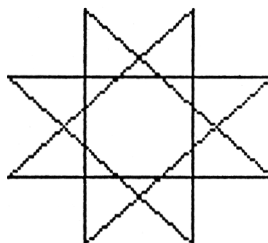






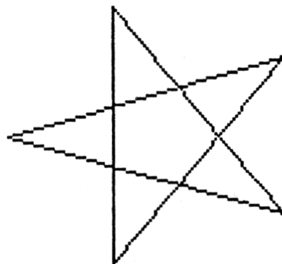




POLIGONO 100 135

figura 14



POLIGONO 110 144

figura 15

Cuando escribes, por ejemplo,

```
POLIGONO 50 90
```

la tortuga avanza 50, gira 90 y se ejecuta el procedimiento POLIGONO, dando valores 50 y 90 a las entradas; esto quiere decir que la

tortuga avanza 50, gira 90 y se ejecuta el procedimiento POLIGONO, dando valores 50 y 90 a las entradas; esto quiere decir que la tortuga avanza 50, gira 90 y se ejecuta el procedimiento POLIGONO, dando valores 50 y 90 a las entradas; esto quiere decir... Bueno, ya es suficiente; como podrás comprobar es una forma cómoda y barata de repetir indefinidamente. Inténtalo con montones de valores de LADO y ANGULO; en algunos casos las figuras son preciosas.

Esta repetición indefinida tiene un inconveniente: un procedimiento que se autorrepite indefinidamente no puede utilizarse para construir otros procedimientos, ya que no se sale de él. No te preocupes: enseguida verás cómo hay que modificar este tipo de procedimientos para que se detengan en el momento adecuado.

Los procedimientos como POLIGONO, que se "llaman" a sí mismos para volver a ejecutarse, reciben el adjetivo "recursivos". Puesto que la recursión es una herramienta importante en programación, vamos a insistir algo más sobre ella.

Veamos un procedimiento que permite escribir números consecutivos en la pantalla, a partir de uno dado:

```
SEA CONTAR :NUMERO
ESCRIBE :NUMERO
CONTAR :NUMERO+1
FIN
```

Cuando escribas CONTAR 1, aparecerá el 1 en la pantalla y se ejecutará CONTAR con la entrada 1+1, es decir, CONTAR 2; éste escribirá el 2 en la pantalla y ejecutará CONTAR con la entrada 2+1, es decir, CONTAR 3, y así sucesivamente hasta que lo pares. Anteriormente dijimos que cada vez que se ejecuta un procedimiento se crean los almacenes privados del mismo. Hagamos la película de lo que ocurre en este caso al escribir CONTAR 1. Utilizaremos "cajas" para representar las variables locales con sus valores almacenados en cada momento:

contar 1

número [1]

escribe 1
contar número + 1 → contar 2

número [2]

escribe 2
contar número + 1 → congar 3

número [3]

escribe 3
contar número + 1 . . .

Observa una cosa: sólo hasta aquí, ya tenemos tres variables con el mismo nombre, NUMERO, y con distintos valores. Cuando se ejecuta la línea ESCRIBE :NUMERO, ¿qué valor de NUMERO se escribe? El correspondiente al almacén privado del procedimiento que se esté ejecutando en ese momento.

Condicional: si..., entonces...

Vamos ahora con la detención del procedimiento. Supongamos que sólo queremos escribir del 1 al 20. A priori sería necesario una instrucción del tipo

Si el número es mayor que 20, entonces párate.

Este tipo de instrucción condicional existe en LOGO, y su sintaxis es la siguiente:

SI ⟨condición que debe cumplirse⟩ [⟨acción que debe realizarse⟩]

que en nuestro caso sería

SI
⟩
ALTO

```
SI :NUMERO > 20 [ALTO]
```

La primitiva "⟩" tiene el significado habitual de "mayor que".

La expresión :NUMERO ⟩ 20 puede ser verdadera o falsa, es decir, tomar los valores VERDAD o FALSO. Si es cierta, LOGO ejecutará lo que esté escrito a continuación entre corchetes, en este caso la primitiva ALTO, que detiene la ejecución del procedimiento en curso.

Las primitivas como ⟩, ⟨, = y otras que más adelante veremos juegan el papel de "predicados", que pueden ser verdaderos o falsos, empleándose por ello en las condicionales.

Podíamos igualmente, haber detenido el procedimiento con la instrucción

```
SI :NUMERO = 21 [ALTO]
```

Obsérvese que la ⟨acción a realizar⟩ si el predicado es VERDAD debe escribirse como una lista, entre corchetes, en forma análoga a la que se usaba en REPITE.

Añadiendo otra entrada para detenerlo, el procedimiento quedaría así:

```
SEA CONTAR :NUMERO :FINAL
SI :NUMERO > :FINAL [ALTO]
ESCRIBE :NUMERO
CONTAR :NUMERO + 1 :FINAL
FIN
```

Verifícalo con diversos valores.

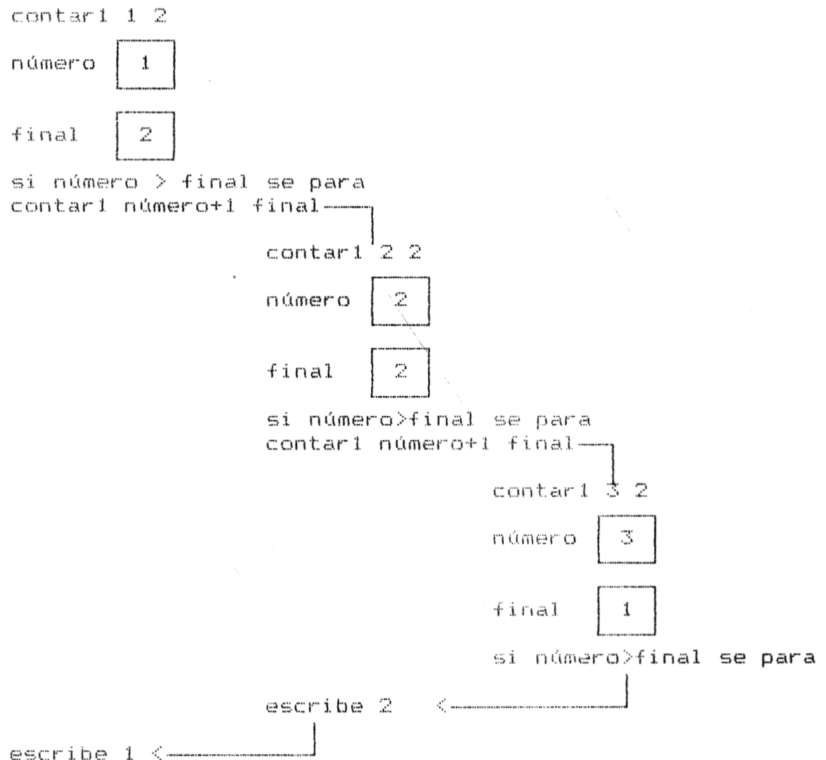
El tipo de recursión que hasta ahora hemos visto puede considerarse como una generalización de la repetición, y suele llamarse "recursión de cola", ya que la llamada recursiva es la última instrucción del procedimiento. Existen otras formas de recursión mucho más potentes, pero también más difíciles de entender. Vamos a hacer una ligera modificación en CONTAR

```
SEA CONTAR1 :NUMERO :FINAL
SI :NUMERO > :FINAL [ALTO]
CONTAR1 :NUMERO + 1 :FINAL
ESCRIBE :NUMERO
FIN
```

Te sugerimos lo siguiente:

1. Piensa con detenimiento, basándote en lo expuesto, qué va a realizar este procedimiento.
2. Ejecútalo y, si los resultados no coinciden con los que esperabas, revisa tus razonamientos.

De todos modos vamos a "poner la película" de CONTAR1 y después analizaremos su marcha. Sigue las flechas.



¿Está claro el esquema? Repasémoslo. La llamada CONTAR1 1 2 crea el almacén privado:

número 1
final 2

A continuación verifica si "1>2"; al ser FALSO el resultado no se ejecuta ALTO. Se llama ahora a CONTAR1 con las entradas 2 y 2, de modo que el CONTAR1 en el que estábamos se abandona momentáneamente. La llamada crea los almacenes:

número 2
final 2

por tanto, ya tenemos dos CONTAR1, cada uno con sus propios almacenes. Como "2>2" es FALSO, se llama a CONTAR1 con los valores 3 y 2, lo que produce los almacenes:

número 3
final 2

y ya tenemos tres CONTAR1 en danza. Hasta el momento LOGO ha realizado mucho trabajo, pero no ha producido ningún resultado observable visible; enseguida se animará el asunto. Como "3>2" es VERDAD, se produce la detención en esta línea y la devolución del control al procedimiento que llamó a CONTAR1 3 2, que es CONTAR1 2 2. Este procedimiento continúa entonces su ejecución en la línea siguiente a la llamada; esto es, ESCRIBE :NUMERO, y como el valor de NUMERO es 2 en el almacén privado de este procedimiento, se escribe el 2. ¡Por fin advertimos algo! Este procedimiento termina, pasando el control a CONTAR1 1 2, que continúa ejecutándose en la línea ESCRIBE :NUMERO, de modo que se escribe un 1, valor de NUMERO en el almacén privado de este procedimiento. Termina así CONTAR1 1 2 y con él toda la historia.

Aunque pueda parecerte complicado, lo ocurrido puede resumirse así:

Cuando un procedimiento, A, llama a otro, B, la ejecución de A se detiene hasta que B haya terminado de realizarse; A continúa entonces ejecutándose a partir de la instrucción siguiente a la llamada.

En particular, cuando B es el propio A —como ocurre en la recursión—, se encuentra uno con varias llamadas al mismo procedimiento coexistiendo simultáneamente, pero con almacenes privados distintos. Destaquemos que la última llamada es la primera en terminar de ejecutarse.

Antes de proseguir con otros ejemplos, una sugerencia: elabora una versión de CONTAR1 que utilice la recursión de cola, es decir, algo así como:

```
SEA DESCONTAR :NUMERO :FINAL
SI :NUMERO < :FINAL [ALTO]
ESCRIBE :NUMERO
DESCONTAR <¿que se pone aquí?>
FIN
```

de modo que la llamada DESCONTAR 10 0 escriba los números 10,9,8,7,6,5,4,3,2,1,0. Por cierto, cuando la "cuenta atrás" llega a 0, suele despegar un cohete. Si quieres...

2.1 Un ejemplo recursivo: los árboles

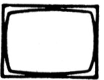
Existen situaciones donde la definición recursiva de un objeto se impone por sí misma. Considera el árbol binario (dos ramas en cada nudo) de la figura 16.

Si quieres describirlo en forma secuencial deberías escribir algo así como:

```
DERECHA 45 AVANZA 60 DERECHA  45 AVANZA 30 DERECHA 45 AVANZA
15 RETROCEDE 15 IZQUIERDA 90
```

etcétera.



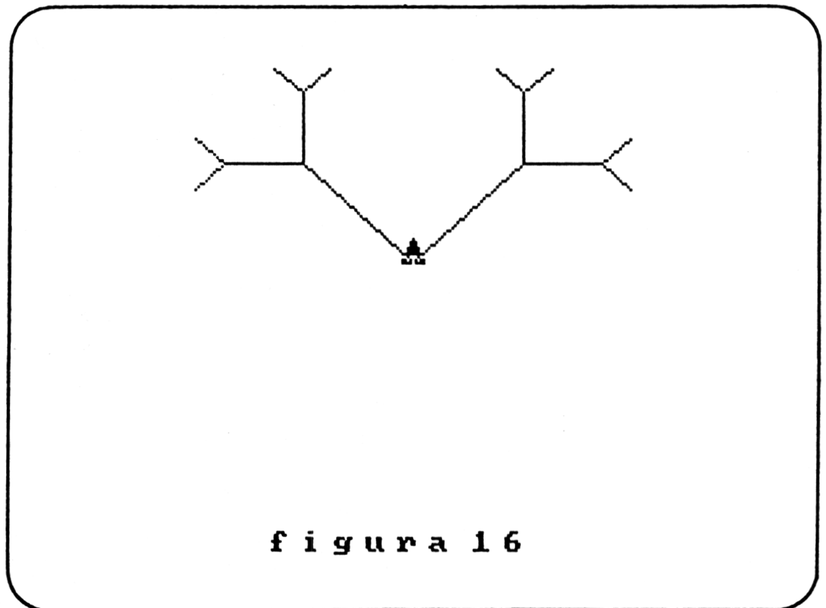



figura 16

Pese a la desmesurada longitud de la definición, no sirve para árboles que tengan más o menos ramas que éste. Tal dificultad es propia de los árboles. La mejor definición es de tipo recursivo:

Un árbol es un par (o más) de ramas en cada una de las cuales hay un árbol.

Según esto:

```
SEA ARBOL :LONG
IZQUIERDA 45
AVANZA :LONG
<aquí el otro árbol>
RETROCEDE :LONG
DERECHA 90
AVANZA :LONG
<aquí el otro árbol>
RETROCEDE :LONG
IZQUIERDA 45
FIN
```

Si queremos que la longitud de las ramas siguientes sea la mitad de las anteriores, basta con escribir, en lugar de ⟨aquí el otro árbol⟩, la llamada ARBOL :LONG / 2.

Ejecuta el procedimiento y comprueba que no se dibuja nada, debido a que no existe ninguna condición para terminar la recursión. Si vuelves a la definición recursiva de árbol, observarás que no refleja exactamente la figura dibujada. La definición completa sería:

Un árbol es un par de ramas en cada una de las cuales hay un árbol o no hay nada.

"No hay nada" significa "no dibujar nada", que equivaldría a la instrucción:

```
SI :LONG < 4 [ALTO]
```

Termina de escribir la versión definitiva de ARBOL, incorporando, en el lugar adecuado, esta última instrucción.

Hemos de advertir que lo efectuado en ARBOL, y también en otros procedimientos, en cuanto a la recursión se refiere, es lo más frecuente. Lo primero es diseñar el procedimiento recursivo y después hay que preocuparse de pararlo.

¿No puede dibujarse un árbol en el que todas las ramas tengan idéntica longitud? En principio bastaría con sustituir las llamadas recursivas, ARBOL :LONG / 2 por ARBOL :LONG. Compruébalo y observa que no funciona, porque hemos modificado algo que afecta a la condición de terminación. En la versión "buena" la longitud de la rama va disminuyendo, por eso se detiene el procedimiento en un momento dado; pero en la "mala" la longitud no disminuye. Si quieres ramas de la misma longitud vas a tener que cambiar la condición de terminación, y una posible solución es tener en cuenta la "frondosidad" del árbol. Al árbol de la figura podríamos asignarle una

"frondosidad" de 3. Tiene dos ramas (frondosidad 1), de las cuales nacen otras dos (frondosidad 2); de éstas sale otro par (frondosidad 3), y se acabó el árbol. Utilizando ese parámetro el título del procedimiento podría ser

```
SEA ARBOL.F :LONG :FROND
```

las llamadas recursivas podrían ser

```
ARBOL.F :LONG :FROND + 1
```

y la condición de terminación

```
SI :FROND > 3 [ALTO]
```

Prueba el procedimiento escribiendo ARBOL.F 40 1.

El inconveniente de este procedimiento es que si deseas uno de frondosidad 4 tienes que editar y modificar la condición de terminación, cambiando el 3 por un 4. Obviamente podrías introducir una nueva variable, FINAL, usando la condición

```
SI :FROND > :FINAL [ALTO]
```







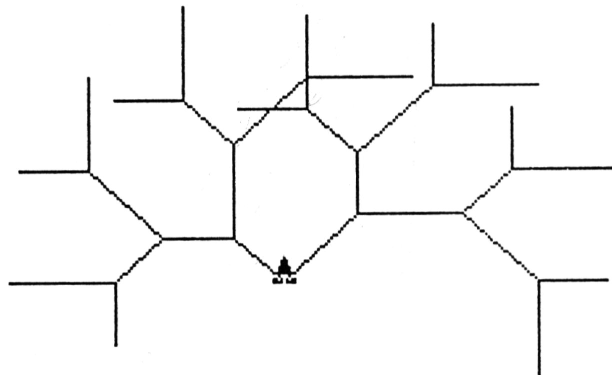


figura 17

Sin embargo, tienes otra solución, y fíjate bien en ella porque a veces sirve para ahorrar variables: consiste en contar al revés, es decir, utilizar como condición de parada

```
SI FROND = 0 [ALTO]
```

tomando como llamadas recursivas

```
ARBOL.F :LONG :FROND - 1
```

Puedes entonces llamar al procedimiento escribiendo ARBOL.F 40 3, si deseas una profundidad 3. Finalmente, ¿podrías conseguir que las ramas izquierdas tuviesen dos tercios de la longitud de las ramas derechas?

Detención de un procedimiento

Recordarás que el POLIGONO recursivo no tenía condición de terminación. Si piensas que, cuando el dibujo de un polígono se termina, la tortuga está en el mismo lugar donde estaba al empezar, su giro total debe haber sido 360 grados o uno de sus múltiplos. Vamos a añadir una nueva variable cuyo valor sea el ángulo que ha girado la tortuga hasta ese momento, y utilizar la primitiva PAUSA —que detiene momentáneamente la ejecución de un procedimiento y permite "preguntar por su salud"— y CO (continúa), que prosigue con la ejecución del mismo. Prueba el procedimiento

PAUSA

```
SEA POLIGONO :LADO :ANGULO :GIRO
PAUSA
AVANZA :LADO
DERECHA :ANGULO
POLIGONO :LADO :ANGULO :GIRO+:ANGULO
FIN
```

con la llamada

```
POLIGONO 50 120 0
```

Se detendrá inmediatamente, ya que PAUSA es la primera instrucción, apareciendo el mensaje

```
PAUSA EN POLIGONO
<La linea del procedimiento en la que está PAUSA>
POLIGONO?
```

CO

En esta situación puedes "preguntar" todo lo que desees. Por ejemplo, cuánto ha girado la tortuga hasta el momento. Para ella teclea ESCRIBE :GIRO, y recibirás como amable respuesta 0. Da la orden CO y verás cómo el procedimiento continúa ejecutándose. En este caso dibujará un lado, girará y volverá a llamar a POLIGONO, produciéndose una nueva pausa. Contestará a ESCRIBE :GIRO con un 120, y si escribes CO continuará el procedimiento. Si quieres puedes seguir escribiendo simplemente CO hasta que el polígono se cierre; entonces pregunta por el valor de GIRO, ya que ese valor es el que te interesa. Deberías probar otros valores de ANGULO para asegurarte cuál es la condición de parada y escribirla en lugar de la instrucción PAUSA, pero para ello es preciso que puedas salir del procedimiento. Vamos a indicarte dos formas de hacerlo.

La primera consiste en dar la orden

```
ENVIA "NIVELSUP
```

con lo que LOGO, obediente, pasa al nivel superior, apareciendo el símbolo "?", que indica que el sistema LOGO está a tus órdenes.

La segunda forma, si conoces la condición de terminación, es la más espectacular. Cuando se produzca la pausa, escribe EDITA "POLIGONO", con lo que te encontrarás metido en el editor; sustituye la línea de PAUSA por la condición de terminación, y sal del editor utilizando la forma habitual. LOGO repetirá el mensaje de pausa; por tanto, escribe CO y espera a que se detenga, ya que el procedimiento que se está ejecutando es el que acabas de definir con la corrección.

Esta habilidad de LOGO para modificar un procedimiento que se está ejecutando, en la forma en que se ha hecho, lo diferencia notablemente de la mayoría de los lenguajes de programación actuales.

Es posible que la primera vez que escribas la condición de terminación el funcionamiento no sea exactamente el que esperabas, en cuyo caso puedes verte obligado a utilizar la solución heroica; no te preocupes, utilízala, corrige y... ¡adelante!

Almacenando procedimientos: archivos

Como más adelante insistiremos sobre este tema, conviene que almacenes el procedimiento en disco para no tener que volver a teclearlo. Antes de hacerlo escribe ECTS (escribe títulos) para ver cuántos procedimientos tienes almacenados en la memoria central. Supongamos que la respuesta sea

```
SEA ARBOL :LONG :FROND
SEA POLIGONO :LADO :ANGULO
```

Si escribes, por ejemplo:

GUARDA

```
GUARDA "VARIOS
```

LOGO, tras una cierta actividad en el disco, enviará el mensaje

```
2 PROCEDIMIENTOS GUARDADOS
```

¿Está claro lo que ha ocurrido? ¿No? Escribe

DIR

```
DIR
```

y verás cómo LOGO muestra en pantalla una relación de los archivos almacenados en el disco, uno de los cuales será ahora

`VARIOS.LOGO` (en APPLE)
`VARIOS.LF` (en IBM)

Lo que LOGO ha hecho ha sido guardar todo lo que sabía, esto es, lo que hay en la memoria central (dos procedimientos en este caso), en el archivo de nombre VARIOS, añadiendo por su cuenta los sufijos LOGO o LF para distinguirlos de otros que no haya almacenado. ¿Quiere esto decir que cuanto había en la memoria central ha pasado al disco y, por tanto, ya no está en ella? Pues no. Comprueba, utilizando ECTS, que los dos procedimientos permanecen. Al disco ha pasado una copia literal de lo existente en la memoria central.

Puesto que sólo deseas guardar POLIGONO, vamos a borrar ARBOL de la memoria central. Para ello escribe

BORRA

```
BORRA "ARBOL
```

y comprueba con ECTS que ARBOL ha desaparecido de la misma y sólo queda en ella POLIGONO. Escribe

```
GUARDA "POLIGONO
```

y verás el mensaje

```
1 PROCEDIMIENTOS GUARDADOS
```

Ya tienes en el disco una copia de POLIGONO almacenada en el archivo POLIGONO. Vamos a guardar también ARBOL, él solo, en un archivo. Este procedimiento no está en la memoria central, pero sí en el archivo VARIOS. Escribe

BOTODO

```
BOTODO
```

(borratodo), con lo cual la memoria central queda "en blanco"; le has "lavado el cerebro" a LOGO, que ya no conoce ninguno de los procedimientos que había "aprendido" (compruébalo mediante ECTS). A continuación da la orden:

TRAE

```
TRAE "VARIOS
```

y verás signos de actividad en el disco. Lo que está haciendo LOGO es pasar una copia de VARIOS a la memoria central. Cuando cesa la actividad, aparece el símbolo "?" para indicar que ya está listo para recibir órdenes, pero en este caso no hay ningún mensaje. Puedes comprobar que existen dos procedimientos en la memoria (con ECTS) y que VARIOS continúa en el disco (con DIR); es decir, lo que ha pasado a la memoria es una copia de lo que había en el disco.

Suprime ahora POLIGONO con BORRA "POLIGONO y guarda ARBOL en el disco con GUARDA "ARBOL. Realizada esta operación, ya no necesitas VARIOS; por tanto, puedes eliminarlo del disco. Escribe

BOARCHIVO

```
BOARCHIVO "VARIOS
```

y comprueba con DIR que VARIOS ha desaparecido del disco.

Si deseas verificar que las copias no se distinguen de los originales, trae ARBOL o POLIGONO y obsérvalos mediante EDITA ⟨nombre⟩ o ECP ⟨nombre⟩.

Un consejo: si te pones a trabajar donde lo ha hecho otra persona, lo primero que debes hacer es usar BOTODO para dejar la memoria "limpia".

Otra forma de guardar diversos procedimientos en distintos archivos es la de empaquetar procedimientos. Supón que tienes en la memoria ARBOL y POLIGONO (si está limpia, para seguir el ejemplo trae ARBOL y POLIGONO del disco). Para empaquetar procedimientos se usa la primitiva PAQUETE; así

PAQUETE

```
PAQUETE "POLI [POLIGONO]
```

Como ves, esta primitiva tiene dos entradas: la primera es el nombre que le das al paquete, POLI, y la segunda es una lista de todos los procedimientos que vas a poner en él, en este caso sólo POLIGONO. Por supuesto el nombre del paquete puede coincidir con el de uno de los procedimientos empaquetados, es decir, podrías haber escrito.

```
PAQUETE "POLIGONO [POLIGONO]
```

Ahora puedes guardar el paquete en el disco con el nombre de archivo que desees. Escribe

```
GUARDA "POLI1 "POLI
```

Puedes comprobar que en el disco se encuentra ahora el archivo POLI1, que contiene el paquete POLI, en el cual está POLIGONO. Si quieres, y es recomendable, puedes usar el mismo nombre tanto para el archivo como para el paquete. Cuando GUARDA lleva dos entradas, como en este caso, la primera es el nombre que va a tener el archivo y la segunda el que ya tiene el paquete que se va a guardar en él.

Realizada esta operación, eliminamos de la memoria el paquete.

```
BOTODO "POLI
```

que borra todo lo referente al POLI (compruébalo con ECTS), y ya podemos guardar tranquilamente ARBOL. Si, tras un BOTODO, quieres introducir en memoria el procedimiento POLIGONO, basta con teclear TRAE "POLI (si escribiste GUARDA "POLI "POLI).

Si guardas o introduces procedimientos, posiblemente te encuentres con dos mensajes.

Al guardar puede aparecer

```
EL ARCHIVO <nombre> YA EXISTE
```

En tal caso LOGO no guarda tu archivo. Puedes optar, según los intereses del momento, por borrarlo del disco y guardar la nueva versión:

```
GUARDA "OTROS
EL ARCHIVO OTROS YA EXISTE
BOARCHIVO "OTROS
GUARDA "OTROS
```

o por cambiar el nombre para guardar las dos versiones:

```
GUARDA "OTROS
EL ARCHIVO OTROS YA EXISTE
GUARDA "OTROS1
5 PROCEDIMIENTOS GUARDADOS
```

Al traer puede aparecer

```
EL ARCHIVO <nombre> NO EXISTE
```

No te preocupes, lo más probable es que te hayas equivocado al teclear el nombre; si es así, corrige y vuelve a intentarlo. Si el mensaje reaparece, usa DIR para saber cuáles son los archivos que tienes en tu disco.

Otras primitivas

Existen algunas primitivas más, referentes a la memoria central, que pueden resultar útiles:

ECTODO

ECTODO muestra en la pantalla todo lo que tengas almacenado en la memoria. Como pasa a una velocidad mayor de la que puedes leer, disponte a pulsar CTRL-W (en APPLE) o CTRL-NUM LOCK (en IBM), para que la pantalla quede congelada y puedas leer tranquilamente (pulsando una tecla cualquiera continúa la escritura en pantalla).

ECTODO "POLIGONO, siendo éste el nombre de un paquete, escribe todo lo que hay en él.

ECPS

ECPS escribe todos los procedimientos que haya en memoria.

ECPS "POLIGONO escribe todos los procedimientos del paquete POLIGONO.

BOPS

BOPS borra todos los procedimientos de la memoria.

BOPS "POLIGONO elimina todos los procedimientos que haya empaquetados en POLIGONO.

No creas que ECTODO y ECPS hacen lo mismo, pues en la memoria puede haber cosas que no sean procedimientos: variables que han sido asignadas con la primitiva HAZ, como veremos más adelante.

2.2 Espirales

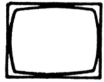
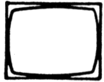
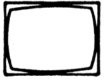
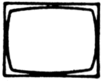
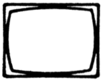
Volvamos a la tortuga. Una sencilla modificación de POLIGONO produce figuras interesantes. Prueba, con valores cualesquiera, el siguiente procedimiento:

```
SEA ESPI1 :LADO :ANGULO
AVANZA :LADO
DERECHA :ANGULO
ESPI1 :LADO + 3 :ANGULO
FIN
```

Si la pantalla se llena de rayas que deshacen el efecto de la espiral, usa VENTANA.

Este procedimiento siempre da el mismo incremento al lado, 3. Escribe ESPI2, con una nueva variable —puedes llamarla INC—, para que puedas modificar el incremento.

ESPI1 y ESPI2 sólo se detienen mediante la "solución heroica", es decir, rompiendo la ejecución con CTRL-G o CTRL-BREAK, según el

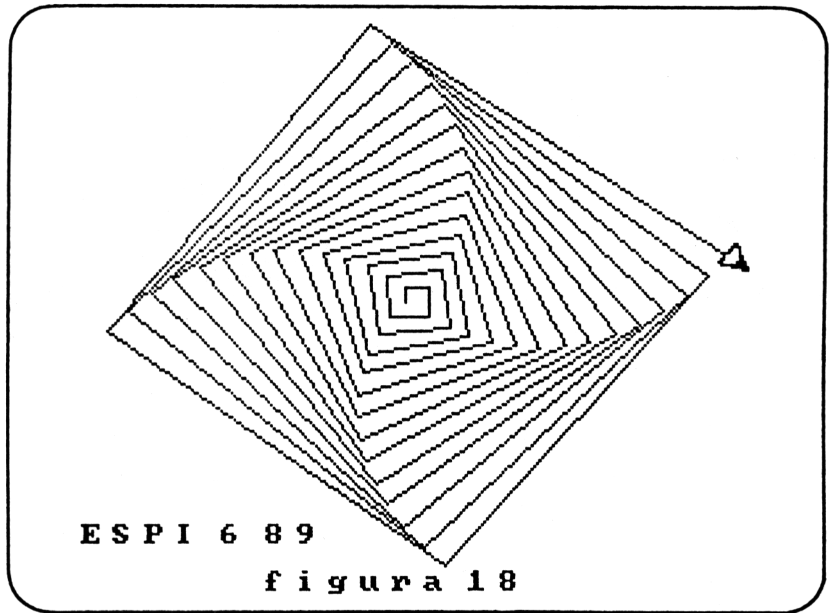


figura 18

caso. Claro está que también puedes definir otro procedimiento, ESPI3, haciendo que se detenga solo. Para ello podrías utilizar la longitud del lado, aunque no es la única posibilidad. ¿Se te ocurre otra? Si ya has escrito este procedimiento, prueba a añadir —tras la llamada recursiva y antes de FIN— la pareja de instrucciones

```
RETROCEDE :LADO
DERECHA :ANGULO
```

con lo que tendrías un ESPI4.

Todas estas espirales "van de lado". Prueba a escribir

```
ESPI5 :LADO :ANGULO :INCANG
```

en la que sólo se incrementa el ángulo. Ejecuta este procedimiento con los siguientes valores:

```
ESPI5 10 1 10
ESPI5 10 10 10
ESPI5 7 3 10
ESPI5 7 5 10
```

Realmente curioso, ¿no? Sería interesante estudiar la relación existente entre los datos del procedimiento y las figuras que produce, no siempre previsibles.

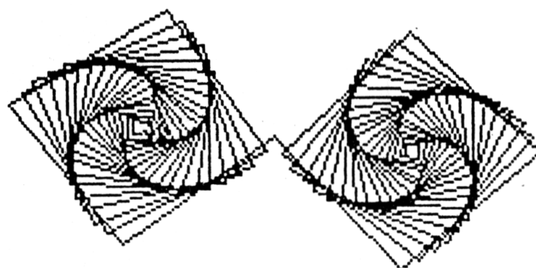










ESPI4 4 92 1 71

figura 19



ESPI5 10 1 10

figura 20

Tras insinuarte que todavía queda “marcha”, ya que puedes incrementar lados y ángulos a la vez (¿ESPI6”), llega el momento de guardar en el disco todas tus espirales.

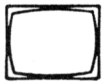
Con el tiempo verás que es fácil olvidarse de lo que hay en un









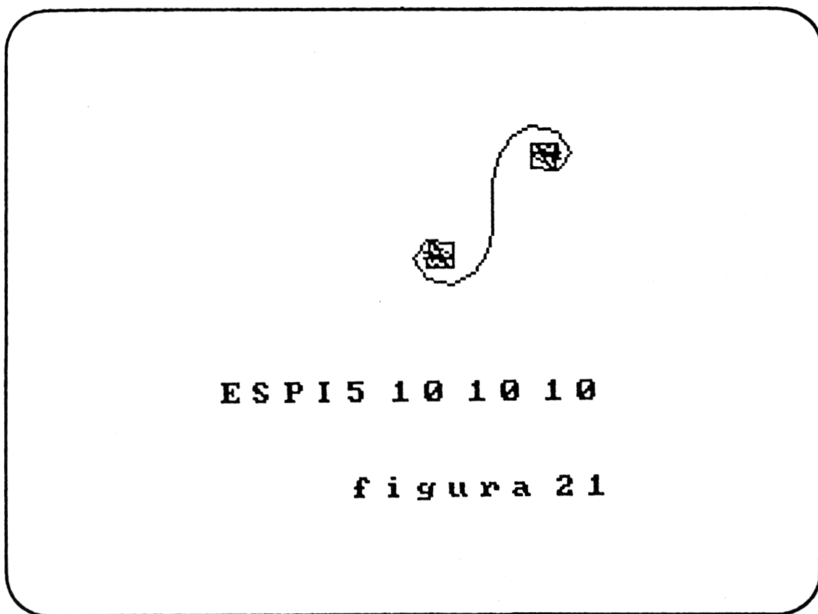


figura 21

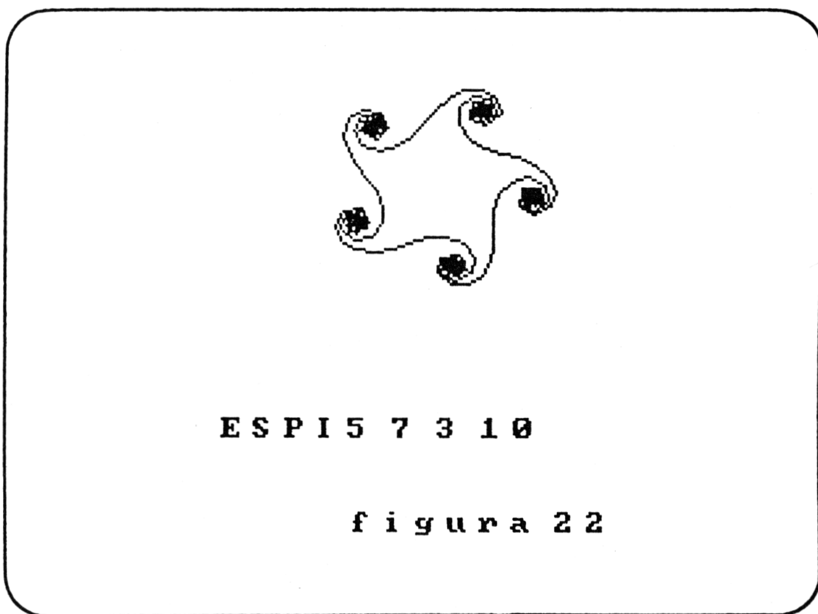


figura 22

archivo, de manera que es conveniente dotarlo de un índice que aparezca en pantalla al traer el archivo a la memoria central. Para ello puedes incluir en el lote, antes de guardarlo, otro procedimiento —puedes llamarlo AYUDA—, que indique lo que hay bajo el nombre ESPI. Es fácil de escribir:







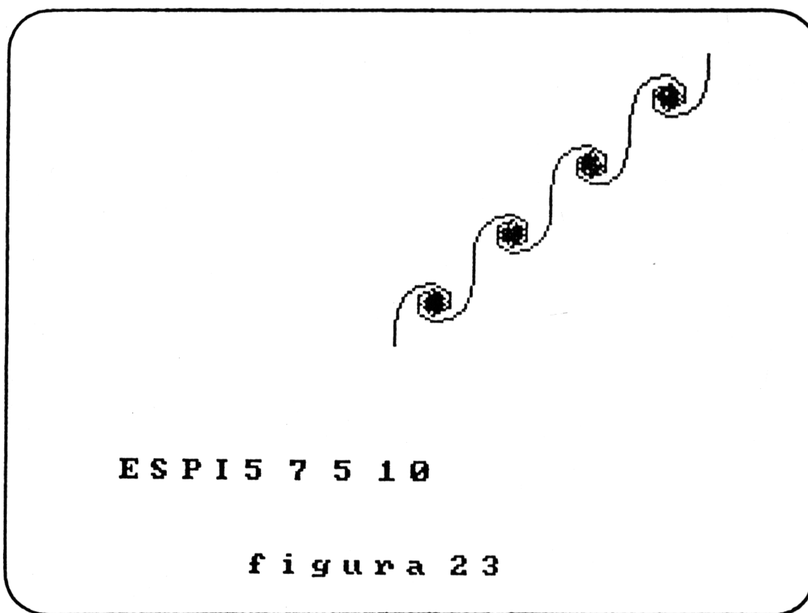


figura 23

```
SEA AYUDA
ECTS
FIN
```

y, al ejecutarlo, te dará los títulos de los procedimientos con sus entradas. Si deseas añadir más aclaraciones en AYUDA es cosa tuya. Pero es preferible que, antes de guardar todos los procedimientos, escribas

ARRANCA

```
HAZ "ARRANCA [AYUDA]
```

ARRANCA es una variable reservada de LOGO a la que con HAZ se le da como valor la lista de procedimientos que sigue (en este caso, uno solo). Por cierto: usa ECTODO y ECPS para observar la diferencia entre ambos.

Si guardas lo almacenado en la memoria, esto es, todos los ESPI, AYUDA, y la variable ARRANCA —cosa que conseguirás escribiendo GUARDA "ESPI—, cuando posteriormente teclees TRAE "ESPI se producirá la ejecución automática de AYUDA, y así sabrás lo que tienes entre manos. De este "arranque automático" es responsable la variable ARRANCA, reservada exclusivamente para tal uso. Pruébalo cuanto antes después de un BOTODO.

ECVS

Si tras la prueba indicada tecleas ECVS (escribe variables), te encontrarás con el mensaje

```
ARRANCA ES [AYUDA]
```

BOVS

y si escribes BOVS (borra variables), ya sabes lo que sucederá. Confírmalo usando ECVS.

Circunferencias y arcos

Seguramente no habrás llegado hasta aquí sin haber comentado algo así como: ¿pero es que "esto" sólo dibuja rectas? ¿No se puede dibujar una circunferencia? La respuesta es: si tú fueras la tortuga, ¿qué harías para dibujar una circunferencia? Piensa un poco. Muévete si es necesario, pero responde antes de continuar.

Efectivamente, la respuesta es avanzar un poco y girar algo. Ya conoces un procedimiento que realiza esta operación, POLIGONO, así que tráelo a la memoria y prueba con POLIGONO 5 5. Inténtalo con varios valores más; por ejemplo, 10 y 10, 15 y 15, 20 y 20, 25 y 25, y todos los que quieras. Observarás que a partir de un cierto número de lados el polígono no se distingue de una circunferencia, debido a la precisión del dibujo.

Habrás observado también que cuando la tortuga ha girado 360 grados ya se ha completado la circunferencia; por tanto, no necesitas efectuar una repetición indefinida; puedes poner, por ejemplo, fijando el angulo de giro en 10 grados:

```
SEA CIRCUN1 :AVANCE
REPITE 36 [AVANZA :AVANCE DERECHA 10]
FIN
```

Prueba con varios valores de AVANCE, a ver qué sucede.







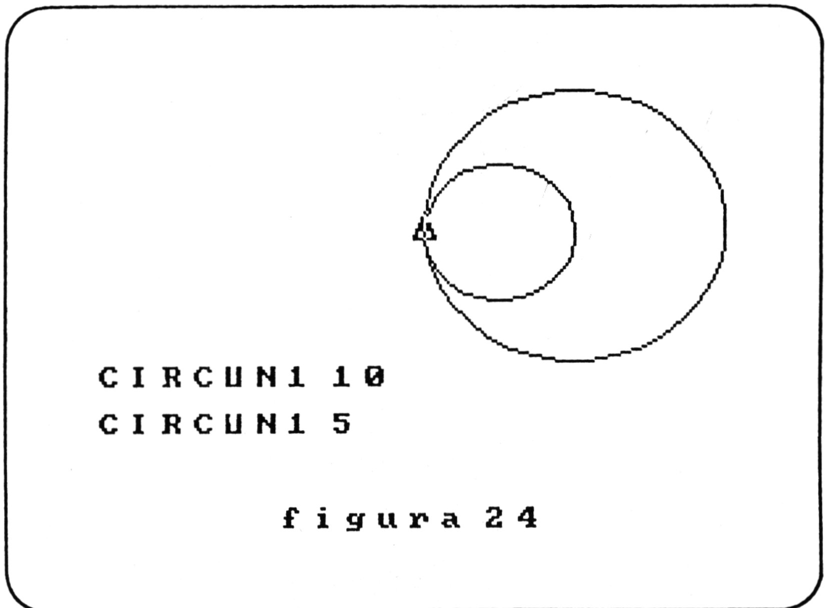


figura 24

Este procedimiento dibuja la circunferencia hacia la derecha, de modo que podrías rebautizarlo como CIRCUN1D. ¿Y si haces CIRCUN1I, que la dibuje hacia la izquierda?

Con los mismos valores de AVANCE que hayas utilizado antes, prueba el procedimiento CIRCUN2D, que puedes escribir sin más que cambiar DERECHA 10 por DERECHA 5 y REPITE 36 por REPITE 72. Superpón un CIRCUN1D y un CIRCUN2D, dando el mismo valor a AVANCE. ¿Tiene una el doble de radio que la otra?

Desde luego, al variar el valor de AVANCE varía el radio de la circunferencia; la cuestión es cómo. Ya que estás acostumbrado a tratar con el radio de una circunferencia, no estaría de más escribir un procedimiento en el que la entrada fuese el radio, en lugar de lo que se avanza; algo así como

```
SEA CIRCUND :RADIO
REPITE 36 [AVANZA <algo> DERECHA 10]
FIN
```

Veamos: en CIRCUN1D la longitud total recorrida por la tortuga era de

36 * :AVANCE

por otro lado, como casi todo el mundo sabe, la longitud de una circunferencia es

2 * PI * :RADIO

por tanto, debe ser

```
36 * :AVANCE = 2 * PI * :RADIO
```

o, lo que es lo mismo,

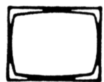
```
:AVANCE = (2 * P I / 36)*:RADIO
```

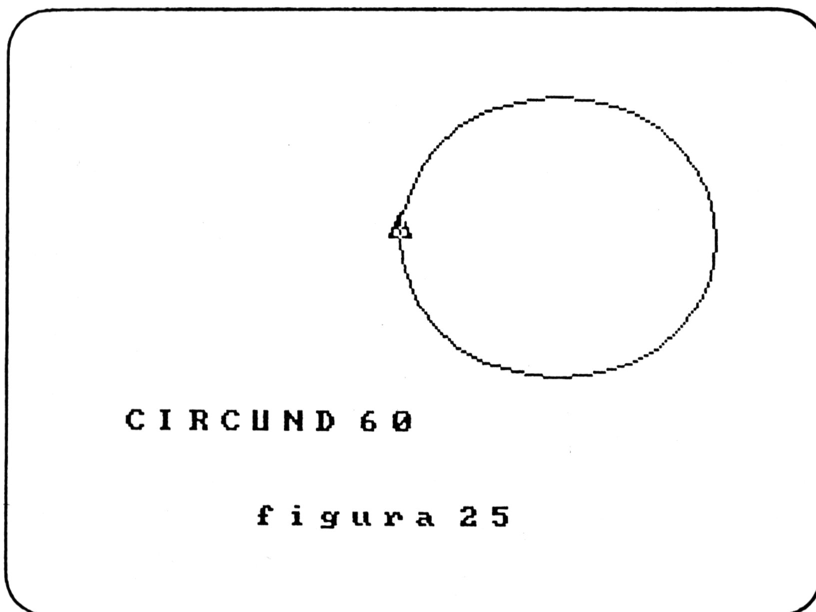
de modo que basta sustituir ⟨algo⟩ por el anterior valor de AVANCE para tener el procedimiento escrito en función del radio.

Observa que de esta manera habría que repetir 36 veces la multiplicación (aunque sustituyas 2 * P I / 36 por 0.1744 hay que multiplicar este valor por :RADIO). Es mejor utilizar una variable local, por ejemplo LONG, escribiendo

```
SEA CIRCUND :RADIO
LOCAL "LONG
HAZ "LONG (2 * PI / 36 ) * :RADIO
REPITE 36 [AVANZA :LONG DERECHA 10]
FIN
```



La primera línea informa a LOGO de que la variable llamada LONG es accesible sólo a este procedimiento (y también a aquellos a los que éste pueda llamar).

La segunda asigna a la variable LONG el valor que se obtenga al realizar las operaciones indicadas; hemos dejado PI, ya que en IBM se dispone de este valor, pero en APPLE deberás sustituir PI por el famoso 3,1416. Las demás líneas no ofrecen pegas para ti. Procediendo en la forma en que lo hemos hecho, la multiplicación se hace una sola vez, con lo cual se gana en rapidez de ejecución.

CIRCUNI está esperando que alguien lo escriba.

Teniendo ambos procedimientos a mano, no debería ser difícil dibujar figuras tales como la 26, 27, 28 y 29.

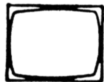
Anímate y efectúalas.

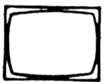
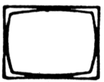
Evidentemente la figura 29 no podrás realizarla sólo con circunferencias: necesitarás un procedimiento que trace arcos, cuya cabecera podría ser

```
SEA ARCOD :RADIO :GRADOS
```

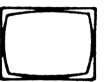
Supón que quieres dibujar un arco de 60 grados; como cada vez la tortuga gira 10 grados, bastaría con que hicieses la repetición 60/10 veces y, en general, :GRADOS 10 veces.

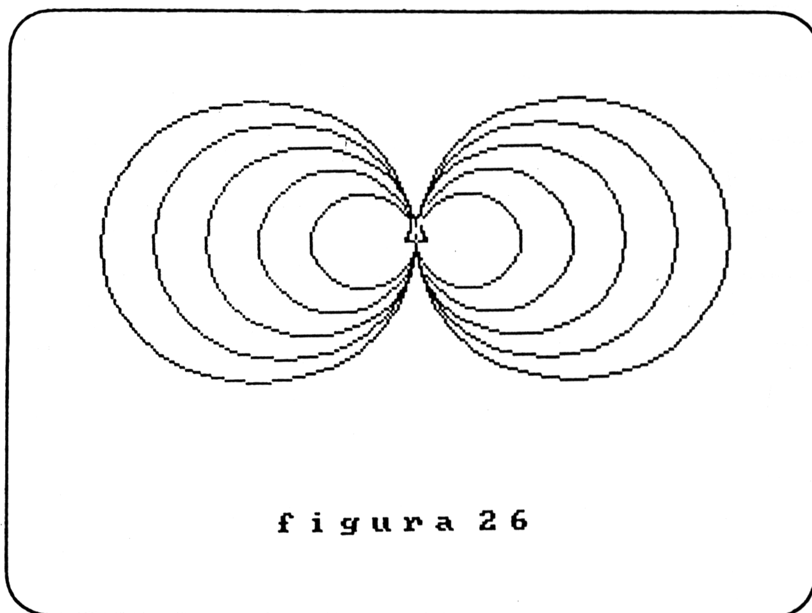
Si el número de grados no es un múltiplo de 10, además de repetir :GRADOS / 10 tendrías que avanzar la parte proporcional a los

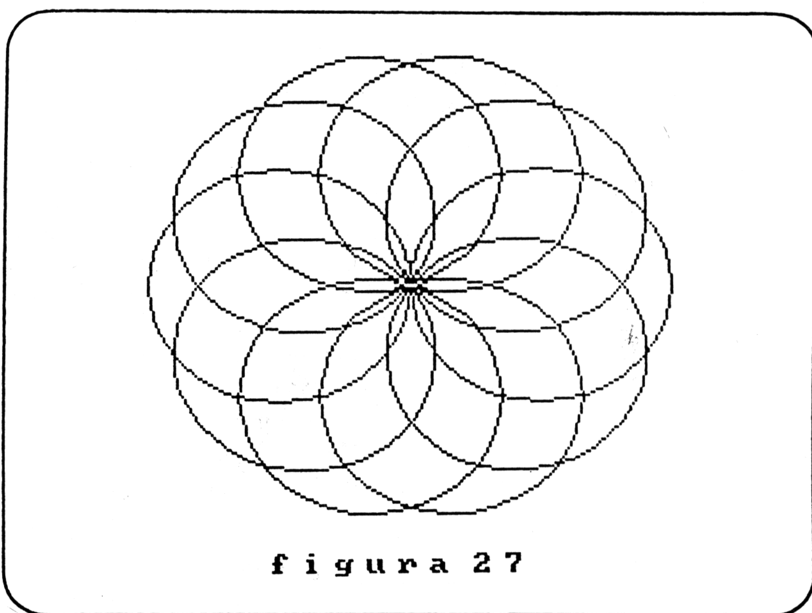





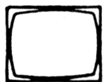
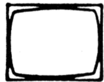
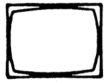
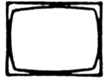



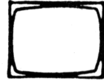
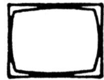
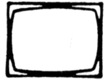


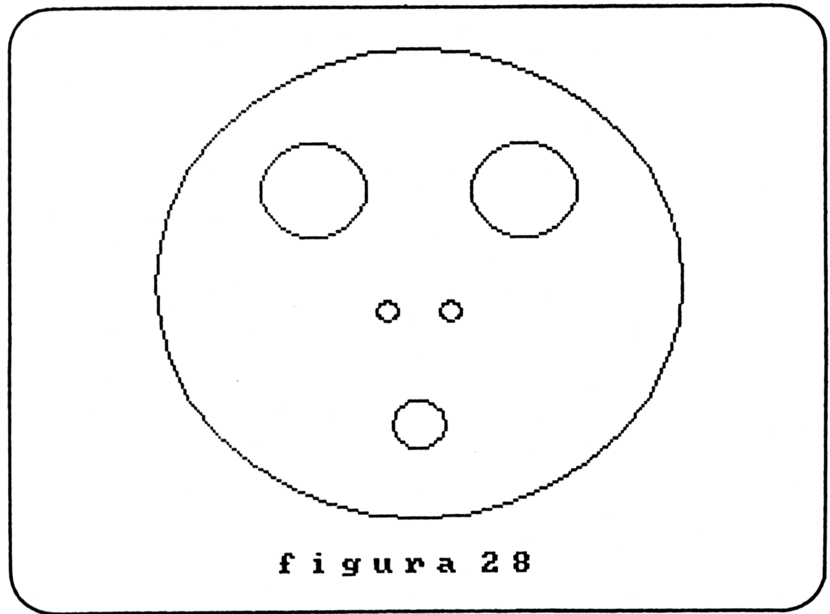


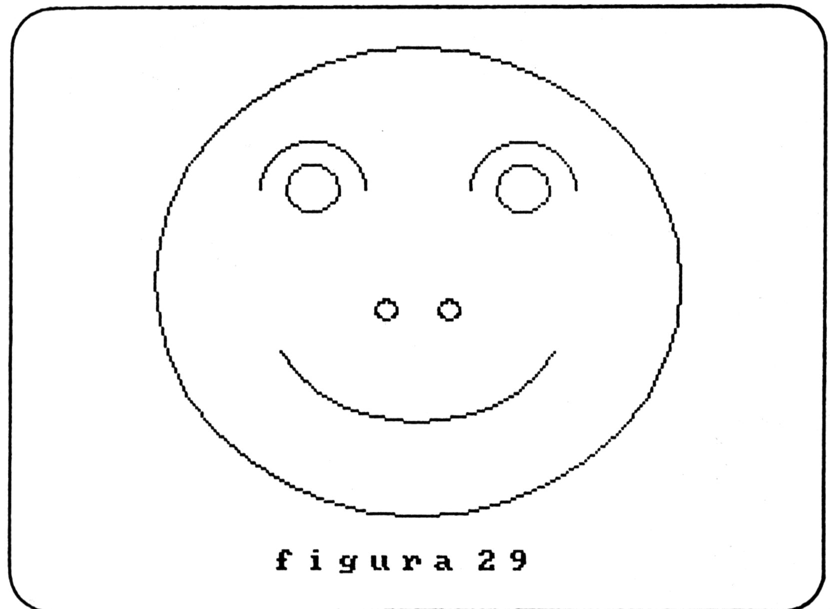
RESTO

grados que faltan y girar estos grados. Los grados sobrantes se obtienen inmediatamente usando la primitiva RESTO, que proporciona el resto de dividir su primera entrada por la segunda; así RESTO 63 10 es 3, RESTO 40 10 0, etc.



COCIENTE
ENT

Para calcular la división entera, en APPLE se usa la primitiva COCIENTE, mientras que en IBM se utilizan dos primitivas, ENT (parte entera) y COCIENTE, que equivale a la división racional en notación prefija, es decir, con el operador delante de los operandos.

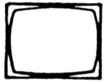
Nosotros vamos a llamar COCIENTE al resultado de la división entera. Si empleas APPLE déjalo igual, y si es IBM escribe ENT COCIENTE. En cuanto conozcas el resto, basta con que avances, como ya hemos dicho, la parte proporcional, esto es (COCIENTE :LONG 10) * :RES, donde la variable RES es el número de grados que faltan, RESTO :GRADOS 10, y gires :RES a la derecha, con lo cual, momentáneamente quedaría así:

```
SEA ARCOD :RADIO :GRADOS
(LOCAL "LONG "RES)
HAZ "LONG (2 * PI / 36) * :RADIO
HAZ "RES RESTO :GRADOS 10
REPITE COCIENTE :GRADOS 10 [AVANZA :LONG DERECHA 10]
AVANZA (COCIENTE :LONG 10) * :RES DERECHA :RES
FIN
```

Observa el paréntesis en la declaración de variables locales: es preceptivo para más de una.

Si no te gusta cómo está el procedimiento —la última línea podría no agradarte—, arréglalo. La última línea sólo debería ejecutarse si el valor de RES es distinto de cero, así que recuerda las condicionales y adelante.

Antes de pedirte que escribas ARCOI, ejecuta el nuevo ARCOD, que habrás escrito para dibujar un arco de 180 grados. ¿No queda un poco asimétrico? Ello se debe a que lo primero que hace la tortuga es







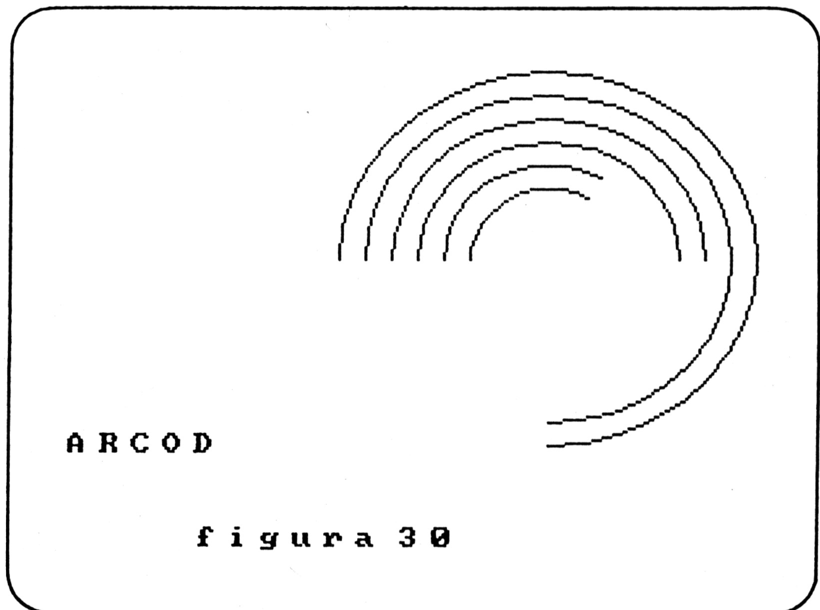


figura 30

avanzar, dibujándose así un trazo recto sin correspondencia en el último tramo. Puedes evitarlo si descompones el giro de 10 en dos de 5, de manera que la lista que se repite quede así:

```
DERECHA 5 AVANZA :LONG DERECHA 5
```

dejando lo demás igual.

Realiza esta corrección en CIRCUND, CIRCUNI, ARCOD y ARCOI y guárdalos todos juntos en el disco para su uso posterior. ¿Recuerdas algo que se ha dicho sobre un procedimiento llamado AYUDA y una variable llamada ARRANCA?

No te quedes sin ejecutar un procedimiento que dibuje la cara sonriente.

Hasta ahora nos hemos limitado a manejar la tortuga «a palo seco», sin apenas ayuda de números ni otros tipos de datos. Pero LOGO puede hacer muchas otras cosas; por tanto, no te pierdas el próximo capítulo de esta apasionante serie, en el que números y listas hacen su entrada triunfal.

ANTE
NTE
ANTECEDEN
LEELISTA
TE = []
CHR$"ANTE
:ANTECED
AND LEE

3
Números y listas

Una de las viejas ideas sobre ordenadores es que éstos son máquinas para calcular rápidamente. Desde luego pueden hacerlo, aunque, como has comprobado, también hacen otras muchas cosas.

Vamos a ver cómo maneja LOGO los números; aquí sí existen diferencias, y grandes, entre APPLE e IBM.

Operaciones

En cuanto a operaciones se refiere, ambos disponen de

SUMA
PRODUCTO

+, −,·*, /, SUMA, PRODUCTO, COCIENTE, RESTO.

Las cuatro últimas son prefijas, esto es, hay que escribir la primitiva antes de sus entradas, como hiciste con COCIENTE y RESTO. Como ya advertimos en su momento, COCIENTE devuelve en APPLE el cociente entero de dos enteros, mientras que en IBM es el equivalente a / en notación prefija.

SUMA y PRODUCTO pueden tener más de dos entradas, en cuyo caso hay que poner la primitiva y sus entradas entre paréntesis.

Si escribes

```
(SUMA 1 2 3 4 5)
```

aparecerá el mensaje:

```
NO SE QUE HACER CON 15
```

Prueba con:

```
ESCRIBE (SUMA 1 2 3 4 5)
```

DIFERENCIA

IBM dispone, además, de la prefija DIFERENCIA. Por ejemplo, ESCRIBE DIFERENCIA 7 2 devuelve un 5.

LOGO dispone de otras funciones:

SEN

SEN devuelve el seno de un ángulo; su entrada es el número de grados.

COS

COS devuelve el coseno de un ángulo; su entrada es el número de grados.

RAC (que ya conoces) devuelve la raíz cuadrada de un número.

IBM ofrece además:

LN

LN devuelve el logaritmo neperiano de su entrada.

EXP

EXP devuelve "e" elevado a su entrada.

POTENCIA

POTENCIA devuelve el resultado de elevar su primera entrada a la segunda; por ejemplo, POTENCIA 2 10 devuelve 1.024.

PI

PI devuelve las cifras del numero π que indique la precisión.

PRECISION

PRECISION permite modificar el número de cifras con las que LOGO va a trabajar y que va a presentar en pantalla.

VALPRECISION

VALPRECISION informa de cuál es la precisión en un momento dado.

Si, al empezar a trabajar, das la orden

```
ESCRIBE VALPRECISION
```

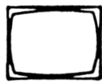
te encontrarás con el número 10, que es la precisión estándar; este valor puede modificarse, pero no puede ser menor que 5 ni mayor que 1.000, aunque algunas primitivas como SEN, COS, LN, EXP, PI, no admiten más de 100. Escribe

```
PRECISION 100
ESCRIBE PI
ESCRIBE EXP 1
```

y sabrás cuáles son las 100 primeras cifras de los números e y π. ¿Las habías visto alguna vez?

Por lo que al LOGO IBM se refiere, admite números escritos







```
3.1415926535897932384626433 83
 2795028841971693993751058209 7
 4944592307816406286208998628 0
 34825342117068

2.7182818284590452353602874 71
 3526624977572470936999595749 6
 6967627724076630353547594571 3
 82178525166427
```

figura 31

como enteros, decimales, en notación exponencial y en notación científica. Por ejemplo:

> 324, 1234567890, como enteros.
> 1.2614, 325.87654 como decimales estándar.
> 1.2345E3, 12.345E2, 123.45E1, todos ellos representantes del número 1234.5, escritos en notación exponencial. 1.2345E3 es lo que en matemáticas escribirías como $1.2345*10^3$, etc.

A la hora de devolver números, LOGO tiene preferencia por el formato decimal; pero si la precisión con la que está trabajando no le permite escribir ninguna cifra significativa, lo escribirá en notación científica, es decir, con una sola cifra entera.

> ESCRIBE 0.0036 produce 0.0036
> ESCRIBE 0.000000000036 produce 3.6E-0011.

No te extrañes del −0011, ya que LOGO puede manejar los exponentes comprendidos entre −9999 y 9999, por lo que el exponente −11 lo escribe como −0011.

Las prestaciones de APPLE en este aspecto son más modestas. Aunque puede escribir y manejar números enteros entre −2147483648 y 2147483648 (aproximadamente 2 elevado a 31), en lo que a exponentes se refiere sólo varía entre −38 y 38, utilizando las notaciones N38 para el primero y E38 para el segundo. La precisión es 7, sin posibilidad de cambiarla.

REDONDEA

Ambos LOGO disponen de un par de primitivas que permiten transformar números reales en enteros. ENT, parte entera de un número, ya la conoces. La otra es REDONDEA, ¿es necesario decir lo que hace?

Prueba ambas con los números 6.4, 6.5, 6.6, −6.4, −6.5, −6.6.

3.1 La media

Vamos con un programa útil: calcular la media aritmética de una serie de números. El programa pedirá los números, que tú irás proporcionando a través del teclado y, cuando le indiques que has terminado, dará el resultado. La ejecución del programa produciría algo así:

```
CALCULO DE LA MEDIA
¿NUMERO?
7
¿NUMERO?
8
¿NUMERO?
3
¿NUMERO?
ADIOS
LA MEDIA ES 6
```

Según esto lo que hay que hacer es:

Escribir el mensaje CALCULO DE LA MEDIA.
Repetir mientras sea necesario:

escribir el mensaje ¿NUMERO?,
leer el número,
actualizar la suma y el número de sumandos.

Escribir el mensaje LA MEDIA ES junto con el resultado de dividir la suma por el número de sumandos.

Vamos a empezar a escribir esto de forma que LOGO pueda entenderlo. Usando las variables SUMA.ACTUAL, SUMANDOS y NUMERO, para almacenar las sumas acumuladas, el número de sumandos y el número que se teclee, podríamos escribir

```
SEA MEDIA
(LOCAL "SUMA.ACTUAL "SUMANDOS "NUMERO)
HAZ "SUMA.ACTUAL 0
```

```
HAZ "SUMANDOS 0
ESCRIBE [CALCULO DE LA MEDIA]
LEE.Y.CALCULA
(ESCRIBE [LA MEDIA ES] :SUMA.ACTUAL / :SUMANDOS)
FIN
```

Entrada de datos por el teclado

LEE.Y.CALCULA se encarga de la repetición. Al no ser conocido el número de veces que hay que repetir, usaremos la recursión. Como pretendemos que la entrada de datos termine al teclear la palabra ADIOS, podemos ya escribir la condición de parada. Tendríamos, pues

PRIMERO
LEELISTA

```
SEA LEE.Y.CALCULA
ESCRIBE "¿NUMERO?
HAZ "NUMERO PRIMERO LEELISTA
SI :NUMERO = "ADIOS [ALTO]
HAZ "SUMA.ACTUAL :SUMA.ACTUAL + :NUMERO
HAZ "SUMANDOS :SUMANDOS + 1
LEE.Y.CALCULA
FIN
```

La primera línea del cuerpo del programa podría haberse escrito también como ESCRIBE [¿NUMERO?].

La segunda es una forma bastante usada para que un programa reciba datos a través del teclado. Cuando LOGO encuentra LEELISTA, hace exactamente eso: "leer" lo que teclees y devolverlo como una lista. Por ejemplo, si tecleas 124 esta primitiva devuelve [124], es decir, la lista con ese único elemento; si tecleas ESTO ESTA CLARO la primitiva devuelve [ESTO ESTA CLARO].

Hemos utilizado la primitiva PRIMERO, cuya entrada es una lista y cuya salida es el primer elemento de la misma. Así pues, el efecto producido por esa línea es asignar a NUMERO el primer elemento de la lista [124], que es 124.

Una advertencia importante: si no tecleas nada, LOGO esperará hora tras hora y día tras día hasta que lo hagas; es decir, no continuará ejecutando instrucciones hasta que no haya leído algo.

Es importante que entiendas cómo se usan los dos puntos para representar el valor de una variable, y cómo las comillas indican que hay que tomar lo que sigue literalmente. Observa cómo se usan ambos signos para hacer que una variable tome el valor que ya tenía, incrementado.

Prueba el programa para ver cómo funciona. Si no has notado ninguna deficiencia, ejecútalo y teclea ADIOS la primera vez que pregunte ¿NUMERO? Te encontrarás con el mensaje

```
NO PUEDO DIVIDIR POR CERO
```

ya que los valores de SUMA.ACTUAL y SUMANDOS siguen siendo el cero inicial. Desde luego es poco probable que alguien ponga en marcha un programa para el cálculo de la media y no le dé ningún dato; pero como puede arreglarse fácilmente y, además, así te presentamos otras primitivas de LOGO, vamos con ello.

Test condicional

Sustituye la última línea del procedimiento MEDIA por

TEST
SIVERDAD
SIFALSO

```
TEST :SUMANDOS = 0
SIVERDAD [ESCRIBE [CALCULAR SIN  DATOS  NO  ES  UNA  DE  MIS
HABILIDADES]]
SIFALSO [(ESCRIBE [LA MEDIA ES] :SUMA.ACTUAL / :SUMANDOS)]
```

TEST admite como entrada una expresión, :SUMANDOS = 0 en este caso. Si la expresión es VERDAD se ejecuta la lista que sigue a SIVERDAD, pero si el valor de la expresión es FALSO se ejecuta la que sigue a SIFALSO. Una forma totalmente equivalente, pero más difícil de leer (para los humanos, no para LOGO) es

```
SI :SUMANDOS = 0 [ESCRIBE [CALCULAR SIN DATOS NO ES  UNA  DE
MIS  HABILIDADES]]  [(ESCRIBE  [LA  MEDIA ES] :SUMA.ACTUAL /
:SUMANDOS)]
```

que es una generalización del condicional que ya viste, y cuya estructura es:

SI ⟨condición que debe cumplirse⟩ [acción a realizar si es verdad] [acción a realizar si es falso]

En lo que sigue utilizaremos una u otra, prefiriendo siempre la que sea más clara de leer.

Filtrado de los datos

Como hay gente para todo, puede ocurrir que si enseñas tu programa a alguien para que se maraville de su perfecto funcionamiento, esa persona teclee SIETE cuando el programa le pida el número. En NUMERO quedará almacenada la palabra SIETE, y cuando se vaya a actualizar la suma recibirás el mensaje

```
A + NO LE GUSTA EL DATO SIETE
```

y LOGO interrumpirá el programa volviendo al nivel superior. La versatilidad de LEELISTA y el que puedas almacenar cualquier cosa

NUMEROP

(número, palabra o lista) en una misma variable son la causa del problema. LOGO supone que tú te preocuparás de que estas cosas no sucedan, y para eso te ofrece primitivas que te permiten "filtrar" las entradas de datos. Una de ellas es NUMEROP, el "predicado de número". Esta primitiva devuelve VERDAD o FALSO, según que su entrada sea o no un número.

Procedimientos que devuelven cosas

Construyamos un procedimiento, al que llamaremos LEENUMERO, que devuelva un número:

```
SEA LEENUMERO
LOCAL "NUMERO
HAZ "NUMERO PRIMERO LEELISTA
TEST NUMEROP :NUMERO
SIVERDAD [DEVUELVE :NUMERO]
SIFALSO [ESCRIBE [ESCRIBE CON CIFRAS, POR FAVOR]]
SIFALSO [DEVUELVE LEENUMERO]
FIN
```

DEVUELVE

La primitiva DEVUELVE permite construir procedimientos que "devuelven" un resultado, como hacen algunas primitivas que ya hemos visto. El efecto de esta primitiva es, por una parte, el mismo que produce ALTO, es decir, interrumpir la ejecución del procedimiento en curso devolviendo el control al procedimiento "padre" (en este caso, LEE.Y.CALCULA), entregándole lo que se indique (en este caso, el valor de NUMERO).

Tal como está el procedimiento, si tecleas un número lo devuelve y termina. Si tecleas otra cosa, escribe el mensaje y devuelve el resultado de ejecutar el procedimiento LEENUMERO, que es un número o bien otra vez el mensaje, y devuelve el resultado de ejecutar el procedimiento LEENUMERO, etc. (¿No sientes cierta inquietud? ¿No? Pues sigue leyendo.)

LEE.Y.CALCULA quedaría así:

```
SEA LEE.Y.CALCULA
ESCRIBE "¿NUMERO? HAZ "NUMERO LEENUMERO
SI :NUMERO="ADIOS ¡RAYOS!
```

Esto no es una primitiva, es una exclamación. LEENUMERO ha filtrado tanto que no va a admitir ADIOS. Si te inquietaste a su debido tiempo tenías razón, y si no fue así, que esto te sirva de lección: a veces, al concentrarse demasiado en los detalles se pierde de vista lo general.

Conviene recordar aquí algo que dijimos sobre los procedimientos; pueden manejarse como "cajas negras", bastando saber cuáles son sus

entradas y qué efecto producen. Quizá es eso lo que se ha perdido de vista en este caso: el procedimiento debía devolver un número o ADIOS, y al diseñarlo hemos pensando sólo en el número.

La disyunción: esto o aquello...

La situación no es grave, no hay que tirarlo todo para volver a empezar. Sólo hay que modificar el test; ahora sería

El valor de NUMERO es un número o la palabra ADIOS que, escrito en LOGO, sería

O

```
TEST O NUMEROP :NUMERO :NUMERO = "ADIOS
```

Como ves, se usa la primitiva O, el "o lógico", en notación prefija (a la que ya debes estar casi acostumbrado). En este caso tiene sólo dos entradas, que son las expresiones NUMEROP :NUMERO y :NUMERO = "ADIOS, que pueden ser verdaderas o falsas.

IGUALP

Por cierto, existe la igualdad en forma prefija, mediante el predicado IGUALP, en la forma IGUALP :NUMERO "ADIOS, por ejemplo.

Cuando haya más de dos condiciones a las que aplicar esta primitiva usaremos paréntesis, en la forma:

```
TEST (O <expresión1> <expresión2> <expresión3>)
```

La primitiva O devuelve VERDAD si alguna de sus entradas tiene el valor VERDAD, y devuelve FALSO si todas sus entradas tienen el valor FALSO.

Conjunción: esto y aquello...

Y

Por supuesto, también existe la primitiva Y, el "y lógico", que devuelve VERDAD si todas sus entradas tienen el valor VERDAD, y devuelve FALSO si alguna de sus entradas tiene el valor FALSO. Análogamente, deben utilizarse paréntesis para más de dos entradas.

Ultimos detalles

Ya está todo terminado. El superprocedimiento MEDIA llama a LEE.Y.CALCULA que, además de llamarse a sí mismo, llama a LEENUMERO. Antes de que lo pongas en marcha, una sugerencia: tú sabes, como autor del programa, que escribiendo ADIOS se obtie-

ne el resultado, pero ¿lo sabe otro usuario del programa? Entonces, ¿cómo puede obtener resultados? Es conveniente que el programa dé la necesaria información al usuario; para ello basta con sustituir, en MEDIA, la instrucción:

```
ESCRIBE [CALCULO DE LA MEDIA]
```

por la línea

```
INSTRUCCIONES
```

Obviamente INSTRUCCIONES sería un procedimiento cuya confección te cedemos gentilmente, ya que tiene un par de astucias que conviene que recuerdes y, ¡cómo no!, una nueva primitiva. (Tranquilo: LOGO tiene menos de 250 primitivas, así que ya falta menos.)

LT

```
SEA INSTRUCCIONES
LT
ESCRIBE [CALCULO DE LA MEDIA]
ESCRIBE []
ESCRIBE [CUANDO EL PROGRAMA ESCRIBA ¿NUMERO?]
ESCRIBE [TECLEA UN NUMERO O LA PALABRA ADIOS PARA TERMINAR]
ESCRIBE "
FIN
```

LT es una primitiva sin entradas, cuyo efecto es el de limpiar la pantalla de texto. ESCRIBE [ ] y ESCRIBE " producen el mismo resultado: una línea en blanco, porque escriben una lista y una palabra vacías, respectivamente. Cuando vayas a escribirlo definitivamente decídete por una de las dos. Y nada más, ponlo en marcha a ver qué sucede.

Polígonos

Dijimos anteriormente que volveríamos sobre POLIGONO, y cumplimos nuestra palabra. Decíamos que un modo de detener el procedimiento era controlando el ángulo girado, que es un múltiplo de 360 grados, cada vez que se cierra la figura. La primitiva RESTO nos permite saber de inmediato si el giro realizado es un múltiplo de 360, ya que en ese caso el resto de dividir el valor de GIRO por 360 debe ser 0.

Pero si escribes como condición de parada RESTO :GIRO 360 = 0, te encontrarás con una sorpresa ya anunciada: el procedimiento no hace nada, ya que 0, el primer valor de GIRO, es un múltiplo de 360, aunque no suela pensarse en él.

Puesto que hay que excluir el cero y LOGO no dispone de una primitiva que signifique "distinto", hay que utilizar "no igual", de

NO

modo que la exclusión se consigue con NO :GIRO = 0. Como esta condición debe ocurrir al mismo tiempo que la de múltiplo, la primitiva Y nos vendrá como anillo al dedo. La condición de parada será:

```
SI Y (NO :GIRO = 0) (RESTO :GIRO 360 = 0) [ALTO]
```

donde los paréntesis se ponen para hacer más comprensible el significado, no siendo imprescindibles.

Así que desempolva POLIGONO y déjalo dispuesto, ya que vamos a estudiarlo un poco.

3.2

Más sobre polígonos

Si el valor de ANGULO es 120 sale un triángulo; si es 90, un cuadrado; si es 60, un hexágono; si es 144, un polígono estrellado, un pentágono, etc. Nos planteamos entonces el siguiente problema:

¿Cómo depende el número de lados del valor de ANGULO?

Si llamamos LADOS al número de lados del polígono, debe cumplirse:

:LADOS * :ANGULO = múltiplo de 360

Pero el segundo término puede escribirse como 360 * :VUELTAS, donde VUELTAS indica el número de "vueltas a la circunferencia" que da la tortuga para cerrar el polígono. De modo que:

```
:LADOS * :ANGULO = 360 * :VUELTAS
```

Puesto que de las variables que intervienen la única que se conoce a priori es ANGULO, el problema es buscar un múltiplo de ANGULO que también lo sea de 360, y, como lo que se busca es la primera vez que esto ocurre, ese múltiplo debe ser el menor posible (positivo, ya que 0 es una solución sin interés del problema). Eso es lo que se llama el mínimo común múltiplo, MCM. En cuanto lo conozcamos sabremos los valores de LADOS y VUELTAS, ya que

```
:LADOS = MCM / :ANGULO
:VUELTAS = MCM / 360
```

Si el valor de VUELTAS es 1 el polígono es convexo, y si es mayor será estrellado.

Mínimo común múltiplo

¿Cómo calculamos el mínimo común múltiplo de ANGULO y 360? Un procedimiento sencillo es el tanteo. Supongamos que el valor de ANGULO es 120; si empezamos con 1 como valor de LADOS, podemos construir la siguiente tabla

LADOS .	LADOS * 120	¿Es múltiplo de 36?
1	120	No
2	240	No
3	360	Sí

Ya hemos terminado, 360 es el MCM (previsible, ¿no?). Con un ángulo de 132 grados podría ser más largo, pero para eso está la máquina, así que vamos a escribir un procedimiento que lo calcule por nosotros. De paso vamos a escribirlo para una pareja de números cualesquiera.

```
SEA MCM :A :B
(LOCAL "N "MULTIPLO)
HAZ "N 1
CALCULA :N
(ESCRIBE [EL MCM ES] :MULTIPLO)
FIN
```

Evidentemente CALCULA se encarga de hacer el trabajo que antes hemos hecho a mano para construir la tabla

```
SEA CALCULA :N
HAZ "MULTIPLO :N * :A
SI RESTO :MULTIPLO :B = 0 [ALTO]
CALCULA :N + 1
FIN
```

Ahora no hay ninguna dificultad para decidir sobre el número de lados del polígono y el número de vueltas. Si el ángulo es 132, como el MCM de 132 y 360 es 3.960, el número de lados es 3960/132 = 30, y el número de vueltas es 3.960/360 = 11.

Cuando se habla de polígonos generalmente se piensa antes en el número de lados que en el ángulo. ¿Cuál es el valor de ANGULO para un pentágono? Se podría responder con otra pregunta: ¿Lo desea V. I. convexo o estrellado? Volvamos a la relación

```
:LADOS * :ANGULO = 360 * :VUELTAS
```

puesto que ahora el valor de LADOS es conocido, tendremos

```
:ANGULO = (360 * :VUELTAS) / :LADOS
```

Si suponemos que el valor de LADOS es 5, se trata de elegir los valores de VUELTAS.

Para el valor 1 se obtiene un ángulo de 72, el pentágono convexo.

Para el valor 2 se obtiene un ángulo de 144, el pentágono estrellado. Y se acabaron los valores a dar, ya que para el valor 3 se obtiene un ángulo de 216, y un giro de 216 a la derecha produce el mismo efecto que uno de 144 a la izquierda. ¿Qué sucede para el valor 4?

Veamos lo que ocurre si el valor de lados es 10.

Para el valor 1 se obtiene un ángulo de 36, el decágono convexo.

Para el valor 2 se obtiene un ángulo de 72. ¡Es el pentágono convexo!

Para el valor 3 se obtiene un ángulo de 108, el decágono estrellado.

Para el valor 4 se obtiene un ángulo de 144. ¡El pentágono estrellado!

Como para el valor 5 ya da un valor de 180, no seguimos. ¿Por qué?

Recapitulemos un poco. Cuando el valor de LADOS es 5 basta con ir dando valores a VUELTAS hasta que se obtiene un valor de ANGULO mayor o igual que 180, y tenemos todos los pentágonos regulares posibles. Sin embargo, cuando el valor de LADOS es 10 esto no funciona. ¿Por qué? Volvamos a la forma en que hemos calculado los valores de LADOS y VUELTAS conocido el de ANGULO.

Para un valor de ANGULO de 72 obteníamos un MCM de 360 y, por tanto, valores de 5 y 1 para LADOS y VUELTAS, respectivamente; esto es, teníamos

$$5 * 72 = 360 * 1$$

pero resulta que también

$$10 * 72 = 360 * 2$$

y también

$$15 * 72 = 360 * 3$$

etcétera.

De todas las parejas: (5,1), (10,2), (15,3), (20,4), etc., que verifican la relación, la primera es la única que se obtiene con el MCM. En el caso de que el ángulo sea 144 el MCM nos proporciona valores para LADOS y VUELTAS de 5 y 2, pero la relación la verifican también 10 y 4, 15 y 6, etc.

Máximo común divisor

Entonces, si queremos, dado el número de lados, ver cuál es el ángulo de los polígonos, estrellados o no, que resultan tenemos que elegir parejas de valores que no tengan divisores comunes. Para valores pequeños del número de lados esto es inmediato. Por ejemplo, para 15 lados sólo tendríamos que considerar, como valores de VUELTAS, 1, 2, 4, 7, que no tienen divisores comunes con 15. (¿Por qué no consideramos 8, 11, 13 y 14?)

Para valores mayores la cosa puede complicarse. ¿Cuántos polígonos hay de 578 lados y qué ángulos son necesarios para dibujarlos? ¿Es 221 un valor admisible de VUELTAS? Dicho de otro modo: ¿tienen 578 y 221 algún divisor común?

Si lo tienen,

$$\begin{array}{r} 578 \\ -221 \\ \hline 357 \end{array}$$

357 y 221 tienen ese mismo divisor común.

Pero entonces

$$\begin{array}{r} 357 \\ -221 \\ \hline 136 \end{array}$$

136 y 221 tienen ese mismo divisor común .

Y naturalmente

$$\begin{array}{r} 221 \\ -136 \\ \hline 85 \end{array}$$

85 y 136 tienen ese mismo divisor común.

Por tanto:

$$\begin{array}{r} 136 \\ -\ 85 \\ \hline 51 \end{array}$$

51 y 85 tienen ese mismo divisor común.

Luego

$$\begin{array}{r} 85 \\ -51 \\ \hline 34 \end{array}$$

34 y 51 tienen ese mismo divisor común.

Es decir:

$$\begin{array}{r} 51 \\ -34 \\ \hline 17 \end{array}$$ y 34 tienen ese mismo divisor común.

Y también:

$$\begin{array}{r} 34 \\ -17 \\ \hline 17 \end{array}$$ y 17 tienen ese mismo divisor: 17.

En definitiva: 17 es el máximo común divisor (MCD) de 578 y de 221, y 221 no es un valor admisible para VUELTAS.

El método anterior de cálculo del MCD de dos números, debido a Euclides, y que deberías intentar justificar, se puede resumir así:

Cálculo de MCD de A y B

Si A = B el MCD es A.
Si A > B entonces MCD (A,B) = MCD (A − B,B)
Si A < B entonces MCD (A,B) = MCD (B − A,A)

Habrás observado que ésta es una definición recursiva, con condición de parada incluida, de manera que traducirla a LOGO es inmediato.

```
SEA MCD :A :B
SI :A = :B [(ESCRIBE [EL MCD ES] :A) ALTO]
SI :A > :B [MCD (:A - :B) :B] [MCD (:B - :A) :A]
FIN
```

Se puede acelerar el procedimiento si en lugar de hacer restas consecutivas, como se ha hecho para pasar de la pareja 578 y 221 a la formada por 136 y 221, se toma el menor de los dos números y el resto de dividir el mayor por el menor, lo que proporciona la misma pareja en una sola etapa. Por otro lado, como vamos a utilizar MCD dentro de otro procedimiento que nos calcule todos los tipos de polígonos con un número determinado de lados, haremos que, en lugar de escribir el máximo común divisor, lo devuelva como resultado. Entonces

```
SEA MCD :A :B
SI :A = :B [DEVUELVE :A]
SI :A > :B [DEVUELVE MCD :B (RESTO :A :B)] [DEVUELVE MCD :A
(RESTO :B :A)]
FIN
```

¿Verdad que ha quedado precioso? Sólo tiene un ligero inconveniente: no "devuelve" más que un error. Verifícalo y comprueba cuál es. Tendrás que modificar la condición de parada, y para ello lo mejor es volver al ejemplo anterior, efectuando las divisiones y observando qué sucede al llegar al final.

Si los números son primos entre sí, es decir, sin divisores comunes, entonces su máximo común divisor debe ser 1, así que ya podemos disponernos a calcular valores de ANGULO y VUELTAS cuando tenemos el valor de LADOS. Aquí tienes un procedimiento que lo hace y, puesto que se trata de un regalo, estúdialo con cuidado.

```
SEA TIPOS.DE.POLIGONOS :LADOS :VUELTAS
LOCAL "ANGULO
TEST MCD :LADOS :VUELTAS = 1
SIVERDAD [HAZ "ANGULO (360 * :VUELTAS) / :LADOS]
SIVERDAD  [SI  :ANGULO  <  180  [ESCRIBE.RESULTADOS] [ALTO]]
TIPOS DE POLIGONOS :LADOS :VUELTAS + 1
FIN
```

ESCRIBE.RESULTADOS queda a tu cargo; debe producir algo así:

```
ANGULO 36 VUELTAS 1
ANGULO 108 VUELTAS 3
```

para el decágono.

Las llamadas a este procedimiento serían:

```
TIPOS.DE.POLIGONOS 10 1
```

Deberían estudiar algunos polígonos y construirlos. Probablemente se puede modificar el programa para que los dibuje él solo, ¿no es cierto?

3.3

Más sobre MCM y MCD

Abandonemos de momento los polígonos y volvamos al MCD y MCM. ¿Qué relación hay entre sus valores y los números que son entradas en ambos procedimientos? Por cierto, del MCM no tienes escrita ninguna versión que devuelva este número; deberías empezar por ahí. Cuando la tengas construye una tabla en la que figure en cada fila la pareja de números, el MCD y el MCM. Si vas a decir que con la información que posees sería más práctico hacer un programa

que la construya, estamos totalmente de acuerdo: hazlo. Encontrarás una relación que permite calcular el MCM conocidos los dos números y el MCD, que te permitirá reprogramar el MCM de forma más eficiente.

3.4

Suma de números racionales

El MCM de dos números interviene en múltiples cuestiones; por ejemplo, en la suma de dos números racionales. No es difícil hacer un programa que realice esta tarea. Supongamos que el programa se porta así, una vez dadas las instrucciones:

```
¿NUMERO? 3/4
¿NUMERO? 5/6
LA SUMA ES 19/12
¿OTRA SUMA? (S / N)
```

Puedes empezar entonces por:

```
SEA SUMA.RACIONALES
INSTRUCCIONES
SUMAR
FIN
```

INSTRUCCIONES deberá indicar simplemente cómo hay que escribir los números. SUMAR se encarga de todo el trabajo, y podría ser algo así:

LEECAR

```
SEA SUMAR
(LOCAL "N1 "N2)
HAZ "N1 LEE.RACIONAL
HAZ "N2 LEE.RACIONAL
RESULTADOS
ESCRIBE [¿OTRA SUMA? (S/N)]
SI LEECAR = "N [ALTO] [SUMAR]
FIN
```

Observa cómo aquí hay una nueva forma de despedirse, en lugar del ADIOS que ya hemos utilizado en MEDIA. Utiliza la primitiva LEECAR, que espera a que se pulse alguna tecla, sea la que sea, y cuya salida es el primer carácter escrito. Esto quiere decir que si tecleas N, NO, NOPI, en cuanto reciba la N la devuelve. No produce "eco" en la pantalla, es decir, si pulsas una N, ésta no aparece en la pantalla y el programa termina ipso facto. Tal como está escrita la condición, si pulsas cualquier otra tecla, por ejemplo la barra espaciadora, el programa continúa con otra suma.

Manejo de listas

LEE.RACIONAL es una variante del LEENUMERO que ya has usado, pero donde la despedida se hace de otra forma y cuyo TEST es, evidentemente, distinto. Si usas en él la variable local NUMERO te encontrarás con que tomará valores del tipo [3 / 4]. De manera que para filtrar los datos tecleados se podrá utilizar un TEST parecido a éste:

```
TEST (Y (NUMEROP  PRIMERO  :NUMERO) (NUMEROP ULTIMO :NUMERO)
(PRIMERO SINPRIMERO :NUMERO = "/))
```

ULTIMO

La primitiva PRIMERO ya la conoces, y suponemos que ULTIMO te la imaginas.

SINPRIMERO

SINPRIMERO es una primitiva cuya entrada es una lista y cuya salida es la misma lista sin el primer elemento. De modo que si :NUMERO es [3 / 4], entonces

```
PRIMERO :NUMERO es 3
ULTIMO :NUMERO es 4
SINPRIMERO :NUMERO es [/ 4]
PRIMERO SINPRIMERO :NUMERO es /
```

Si el test te parece adecuado, ya tienes prácticamente hecho LEE.RACIONAL. Termínalo.

SINULTIMO

Hay otra primitiva, SINULTIMO, cuya definición adivinas, sin duda.

Con las cuatro primitivas, PRIMERO, ULTIMO, SINPRIMERO, SINULTIMO, puedes manipular tranquilamente las listas. El segundo elemento de una lista es el primer elemento de la lista sin el primero; el tercero es el primero de la lista sin el primero de la lista sin el primero, etc.

ITEM

Esta notación puede hacerse a veces muy pesada, de modo que LOGO proporciona otra primitiva, ITEM, que devuelve directamente un elemento de una lista. Sus entradas son un número y una lista, y su salida es el elemento de la lista que ocupa el lugar indicado por el número. Siguiendo con el ejemplo anterior,

```
ITEM 1 :NUMERO es 3
ITEM 2 :NUMERO es /
ITEM 3 :NUMERO ES 4
```

Todas estas primitivas funcionan también con palabras, en lugar de listas, del modo que puedes suponer: PRIMERO devuelve la primera letra de la palabra; SINPRIMERO devuelve la palabra sin la primera letra, etc. ITEM, en el APPLE, sólo funciona con listas.

Si no quieres modificar el test usando ITEM y sigues estando seguro de él, continuamos.

RESULTADOS deberá hacer los cálculos y escribir el resultado. Le vendrá muy bien la variable local DENOMINADOR, para almacenar éste. Y como el denominador común se calcula con el MCM de los denominadores, haremos:

```
HAZ "DENOMINADOR MCM ULTIMO :N1 ULTIMO :N2
```

Sólo queda sumar los numeradores de las fracciones equivalentes a las dadas, que tienen por denominador el valor de DENOMINADOR; esto es,

```
(ESCRIBE [LA SUMA ES] SUMA <numerador 1>  <numerador  2>  "/
:DENOMINADOR)
```

donde <numerador 1> es PRIMERO :N1 * (:DENOMINADOR / ULTIMO :N1) y una cosa análoga para <numerador 2>.

Si lo deseas puedes eliminar el procedimiento RESULTADOS, escribiendo en su lugar, en SUMA, las dos instrucciones que lo constituyen. Sea como sea, prueba el programa y, si has dejado el test que había en LEE.RACIONAL, dale estos "números"

3 / 0 y 1 / 4

Desde luego el primero no es un número racional, pero el filtro se lo va a "tragar". Ya tienes una modificación que hacer.

Prueba ahora con la parejita

0.3 / 1.2 y 1.3 / 2.7

¿Qué hará el MCM? NUMEROP te dice si una entrada es un número, pero no te garantiza que sea un entero. Si recuerdas la primitiva ENT, parte entera, ENT 1.2 es 1, ENT 1 es 1. ¿Te sugiere algo?

El programa, como todos, puede mejorarse. Un consejo: no te empeñes nunca en lograr una versión inmejorable de un programa; no existe. Piensa las cosas detenidamente y perfecciona progresivamente los programas, como hemos hecho hasta ahora. No te preocupes excesivamente por los filtros: un programa "comercial" debe tenerlos, pero eso puede hacerse en versiones posteriores.

No obstante, todavía hay algo que conviene decir, y que pertenece al epígrafe de "piensa las cosas detenidamente". El programa no simplifica el resultado, ni tampoco los datos; debería hacerlo, y no resulta muy difícil si se dispone del MCD, ya que basta dividir numerador y denominador por él para conseguir la simplificación. Anímate.

3.5
Inversión de una frase

Se pueden hacer muchas más cosas con las listas. Por ejemplo, un programa que reciba una frase y la devuelva en orden inverso. Así:

```
¿FRASE? DABALE ARROZ A LA ZORRA EL ABAD
ABAD EL ZORRA LA A ARROZ DABALE
```

Mediante el uso de listas es sencillo. Una vez leída la lista

```
HAZ "FRASE.INICIAL LEELISTA
```

se trata de escribir la última palabra; a continuación la última de la lista sin esa palabra, y así sucesivamente. El procedimiento

```
SEA INV.LISTA :FRASE.INICIAL
ESCRIBE ULTIMO :FRASE.INICIAL
INV.LISTA SINULTIMO :FRASE.INICIAL
FIN
```

casi hace lo que queremos. Le falta la condición de terminación, que no tiene problemas; elige entre estas dos

VACIAP

```
SI :FRASE.INICIAL = [] [ALTO]
SI VACIAP :FRASE.INICIAL [ALTO]
```

Ambas significan lo mismo:

si la frase es la lista vacía, parar;
si la frase tiene la propiedad de ser vacía, parar.

La otra cuestión es que el resultado lo obtendrías así:

```
ABAD
EL
ZORRA
```

etcétera.

TECLEA

Para que te salga todo en la misma línea debes utilizar la primitiva TECLEA que, a diferencia de ESCRIBE, no pasa el cursor a la línea siguiente cuando termina de escribir, sino que lo deja en la posición siguiente al último carácter escrito.

Para que las palabras queden separadas por un espacio, deberás incluirlo al final de cada una de ellas. La orden CAR 32 es sinónimo de “espacio en blanco”. Luego aclararemos esto.

```
(TECLEA ULTIMO :FRASE.INICIAL CAR 32)
```

Por supuesto, cuando termines de escribir, el cursor quedará un lugar después de la E de DABALE, de modo que deberías añadir un ESCRIBE [ ]. Lo más conveniente es hacer un superprocedimiento, de nombre INVIERTE.FRASE, que llame a INVIERTE.LISTA, y luego ejecute ESCRIBE [ ].

Puede llevarse el asunto más allá si se invierten también las letras de cada palabra. Puesto que vamos a insertarlo en INV.LISTA, lo escribiremos de forma que devuelva la palabra invertida; puede hacerse en forma análoga a la inversión de frase, pero vamos a presentarte otra primitiva. Por ejemplo:

```
SEA INV.PAL :PAL
SI :PAL = " [DEVUELVE "]
DEVUELVE PALABRA (ULTIMO :PAL)(INV.PAL SINULTIMO :PAL)
FIN
```

PALABRA

PALABRA es una primitiva que admite al menos dos entradas y devuelve la palabra formada por ellas. Para más de dos entradas son necesarios paréntesis: (PALABRA “H “O “L “A) devuelve HOLA.

Si colocas INV.PAL en el lugar apropiado de INV.LISTA tendrás un INVIERTE.TODO. Pruébalo.

3.6

Código cifrado

ASCII

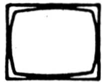
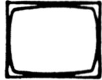
CAR

Existe un código numérico llamado ASCII *(American Standard Code for Information Interchange)* en el que a cada carácter le corresponde un número, desde 0 hasta 127 (LOGO IBM proporciona otros 128 caracteres más), de modo que decir CAR 32 es decir “el carácter número 32 del código ASCII”, que representa el espacio. Análogamente CAR 65 es la A, CAR 90 es la Z, etc. (inténtalo con otros números si lo deseas).

La recíproca de CAR es la primitiva ASCII, cuya entrada es un carácter y cuya salida es el número de código correspondiente; ASCII “A devuelve 65, ASCII “a devuelve 97.

Aquí tienes un entretenimiento; si sustituyes cada letra de una palabra por CIFRADO :LETRA, obtendrás la palabra escrita en código. Utilizando estas primitivas tendrías:

```
SEA CIFRADO :LETRA
LOCAL "NUM
HAZ "NUM (ASCII :LETRA) + 10
SI :NUM > 90 [HAZ "NUM :NUM - 26]
DEVUELVE CAR :NUM
FIN
```




```
?CODIGO

¿Frase a codificar?
ESTO ES UNA PRUEBA DEL PROGRAMA CODIGO

La frase codificada es:
OCDY OC EXK ZBEOLK NOV ZBYQBKWK MYNSQY

?DESCIFRA

¿Frase a decodificar?
OCDY OC EXK ZBEOLK NOV ZBYQBKWK MYNSQY

La frase original era:
ESTO ES UNA PRUEBA DEL PROGRAMA CODIGO
```

figura 32

donde 10 es la “clave secreta” y 90 y 26 se usan para que el carácter obtenido sea una letra comprendida entre la A y la Z.

Como este procedimiento puedes colocarlo en uno que procese todas las letras de una palabra, no forzosamente en orden inverso, y éste a su vez en otro que procese una frase, puedes construir el programa CODIGO, que reciba una frase y la devuelva cifrada, así como un programa que realice la operación de descifrado.

3.7 Capicúas o palíndromos

Volvamos al procedimiento que invierte una palabra. Un número es capicúa si

```
:NUMERO = INV.PAL :NUMERO
```

toma el valor VERDAD.

Considera el siguiente problema:

12 no es capicúa, pero 12 + 21 = 33 sí lo es;
48 no es capicúa, 48 + 84 = 132 tampoco lo es;
132 no es capicúa, pero 132 + 231 = 363 sí lo es.

Dado un número, si es capicúa, hemos terminado; si no lo es le sumamos el número formado por sus cifras en orden inverso, y repetimos si el resultado no es capicúa. ¿Se puede continuar indefinidamente, o siempre se termina llegando a un capicúa? No será difícil elaborar un programa que compruebe sistemáticamente los números con objeto de dar una pista. Antes de empezar practica un poco con lápiz y papel, y no excedas de 1.000 como límite superior; en IBM puedes extenderte más, usando PRECISION. Una advertencia: procura que el programa vaya escribiendo cada uno de los números que investiga para que compruebes que "está trabajando". Y otra: ten en cuenta que en APPLE, si pasaras de 2 elevado a 31, te encontrarías manejando reales y no podrías continuar la investigación.

El procedimiento CAPICUAS utilizaría las siguientes entradas:

NUMERO, para el número que se está investigando.
COTA, para almacenar el límite: 2 elevado a 30.
FINAL, para almacenar el último número que vamos a investigar.

Su esquema de funcionamiento sería:

- Si el valor de NUMERO es mayor que el de FINAL, hemos terminado.
- Escribir el valor de NUMERO.
- Test para saber si el número es capicúa:
  - Si es verdad, escribir que lo es.
  - Si es falso, poner en marcha el procedimiento que va sumando los inversos.
- Pasar al valor siguiente de número.

El procedimiento que va sumando los inversos tendría como entrada el valor de NUMERO en ese momento, y su condición de parada sería la de sobrepasar COTA, en cuyo caso debería emitir el mensaje de "no puedo continuar la investigación" y detenerse. En el supuesto de que en sucesivas llamadas a sí mismo encuentre un capicúa, debe emitir el mensaje oportuno y terminar.

Anímate y escribe el programa.

3.8 Traductor automático

Estudiemos otra aplicación de las listas. Para ello, construiremos un programa que actúe como un diccionario. Puedes elegir los idiomas que quieras; como ejemplo vamos a utilizar un inglés-español. El funcionamiento podría ser el siguiente:

```
¿PALABRA? DOG
PERRO
¿PALABRA? HORSE
CABALLO
¿PALABRA? GRACEFUL
NO LA CONOZCO
¿PALABRA? MAN
HOMBRE
¿PALABRA? ADIOS
```

En primer lugar hay que usar un par de listas, INGLES y ESPAÑOL, en las que pondremos las palabras y sus traducciones. Algo así:

```
HAZ "INGLES [DOG CAT HORSE BOY WOMAN MAN]
HAZ "ESPAÑOL [PERRO GATO CABALLO CHICO MUJER HOMBRE]
```

De esto se podría encargar el procedimiento INICIALIZA, que también podría dar algunas instrucciones si lo juzgas necesario. Vamos a la miga del asunto.

Hay que repetir, hasta que se reciba la palabra ADIOS:

Escribir ¿PALABRA? y "leerla".

Si está en el diccionario entonces escribir su traducción; si no, escribir el mensaje "NO LA CONOZCO".

El uso de TECLEA te vendrá muy bien para la escritura de ¿PALABRA?; en cuanto a leerla, puedes usar

```
HAZ "PAL PRIMERO LEELISTA
```

aunque en IBM puedes utilizar

```
HAZ "PAL LEEPALABRA.
```

Miembro de una lista

El programa ahora ya sabe la palabra que se quiere traducir, pero necesita saber si está o no en INGLES (si fuese ADIOS, el programa habría terminado). Este es un problema que puedes resolver comparando el valor de PAL con la lista INGLES, pero es más sencillo todavía: la primitiva MIEMBROP lo hace directamente. Tiene dos entradas, una palabra y una lista, y devuelve VERDAD si la palabra está en la lista, y FALSO si no lo está. En IBM las entradas pueden ser un carácter y una palabra, o una lista y una relación de listas.

MIEMBROP

Así pues, usando esta primitiva, tendríamos

```
TEST MIEMBROP :PAL :INGLES
SIVERDAD [TRADUCE :PAL]
SIFALSO [ESCRIBE [NO LA CONOZCO]]
```

Puesto que hemos escrito las listas "paralelas", si se localiza el lugar que ocupa la palabra en la lista INGLES, la traducción es simplemente ITEM ⟨lugar⟩ :ESPAÑOL.

Localizar el lugar que ocupa la palabra en la lista es, en cierto modo, la inversa de ITEM; mas, sintiéndolo mucho, vamos a darte una mala noticia: esta vez no hay primitiva y, por tanto, hemos de diseñar el procedimiento LUGAR; obviamente necesita dos entradas, una palabra y una lista, y debe devolver el lugar. Si la palabra es la primera, devolverá 1, y, si no, 1 más el lugar que ocupe la palabra en el resto de la lista; en estas condiciones, cuando la palabra ya no esté en la lista deberá devolver 0, es decir:

```
SEA LUGAR :PAL :LISTA
SI NO MIEMBROP :PAL :LISTA [DEVUELVE 0]
TEST :PAL=PRIMERO :LISTA
SIVERDAD [DEVUELVE 1]
SIFALSO [DEVUELVE 1+LUGAR :PAL SINPRIMERO :LISTA]
FIN
```

La integración de este procedimiento en TRADUCE no ofrece ninguna dificultad:

```
SEA TRADUCE :PAL
LOCAL "LUGAR
HAZ "LUGAR LUGAR :PAL :INGLES
ESCRIBE ITEM :LUGAR :ESPAÑOL
FIN
```

Variables y valores

Existe un modo más sofisticado de traducir aprovechando las posibilidades de asignación de variables en LOGO. Prueba con

```
HAZ "WOMAN "MUJER
ESCRIBE "WOMAN
```

Este resultado ya te lo esperabas.

```
ESCRIBE :WOMAN
```

¿Te lo esperabas? ¿Qué te parece?

Si se asigna a cada palabra en inglés su correspondiente traducción en español (de ello se encargaría el procedimiento ASIGNA), la traducción sería el valor de la variable correspondiente a la palabra leída. Prueba ahora con

```
HAZ "WOMAN "MUJER
HAZ "PAL PRIMERO LEELISTA
```

el signo de interrogación desaparece y queda sólo el cursor, en espera de que escribas algo. Teclea WOMAN. Ahora el valor de PAL, :PAL, es la palabra WOMAN; estamos interesados en el valor de WOMAN, que es la palabra MUJER. La primitiva VALOR te permite obtenerlo. Escribe

VALOR

```
ESCRIBE :PAL
```

y obtienes como respuesta...

```
ESCRIBE VALOR :PAL
```

y obtienes...

Así pues, lo que antes era el procedimiento TRADUCE, que utilizaba el procedimiento LUGAR, se reduce ahora simplemente a

```
ESCRIBE VALOR :PAL
```

de modo que esta parte se simplifica ostensiblemente. Sin embargo, es necesario incluir ASIGNA en INICIALIZA. No es nada complicado, ya que basta con asignar a la primera palabra inglesa la primera española y hacer lo mismo con el resto (SINPRIMERO) de cada lista, hasta que se terminen. Algo semejante a

```
SEA ASIGNA :LISTA1 :LISTA2
SI VACIAP :LISTA1 [ALTO]
HAZ PRIMERO :LISTA1 PRIMERO :LISTA2
ASIGNA SINPRIMERO :LISTA1 SINPRIMERO :LISTA2
FIN
```

A este procedimiento lo llamaremos desde INICIALIZA así:

```
ASIGNA :INGLES :ESPAÑOL
```

3.9 ¿Programas que aprenden?

Quizá pienses que esto es mucho programa para un diccionario tan modesto. Tienes razón. Pero es posible introducir unas cuantas palabras y enseñarle las demás. ¿Puede aprender un programa? Hermoso tema para una discusión. ¿Por qué no la organizas en serio?

Seamos modestos: sólo pretendemos que cuando el programa no sepa una palabra su respuesta sea:

```
NO LA CONOZCO. ¿QUE SIGNIFICA?
```

y entonces debes teclear su traducción.

El programa la incorporará a sus listas, de modo que la próxima vez sí la sabrá. No es muy difícil, pero antes de continuar repasemos brevemente en qué situación podría encontrarse el programa; probablemente así:

```
SEA TRADUCTOR.X
INICIALIZA
TRADUCIR.
FIN

SEA TRADUCIR
LOCAL "PAL
(TECLEA [¿PALABRA?] CAR 32)
HAZ "PAL PRIMERO LEELISTA
SI :PAL = "ADIOS [ALTO]
TEST MIEMBROP :PAL :INGLES
SIVERDAD [<traducir la palabra>]
SIFALSO [ESCRIBE [NO LA CONOZCO]]
FIN
```

Hemos puesto TRADUCTOR.X porque puedes tener dos versiones, que diferirán en el procedimiento INICIALIZA (en una de ellas éste llamará a ASIGNA) y en la forma de traducir la palabra.

Aparentemente lo único que debe modificarse es la parte de SIFALSO; ésta debería ser un procedimiento que realice lo siguiente:

Leer la traducción que se le proporciona, almacenándola en PALTRADUCIDA (recuerda que a procedimientos y variables puedes ponerles los nombres que quieras), y efectuar una de estas dos operaciones, según la versión: o asignar PALTRADUCIDA a PAL, incorporando ésta a la lista INGLES, o colocar PAL en la lista INGLES y PALTRADUCIDA en la lista ESPAÑOL.

Construyendo listas

PONPRIMERO
PONULTIMO

De todo lo expuesto, lo único que puede tener alguna dificultad para ti es lo de colocar una palabra en una lista, pero ésta desaparece usando una de las primitivas PONPRIMERO o PONULTIMO. Ambas tienen dos entradas (una palabra y una lista) y una salida (una nueva lista en la que la palabra figura en primero o último lugar, según la primitiva usada). Esto elimina el problema, y ya puedes escribir el programa TRADUCTOR versiones 3 y 4.

Este programa aprende (¿?), pero no recuerda durante mucho tiempo, sólo mientras está ejecutándose. Claro está que si lo deseas puede tener una memoria de elefante: basta con que guardes en el disco lo que haya aprendido, de modo que en la siguiente ocasión que lo utilices "recuerde" todo.

Veamos qué problema puede surgir para poner en práctica esta recomendación. Si has utilizado, por ejemplo, TRADUCTOR.3, y escribes GUARDA "TRADUCTOR.3, las variables permanecen almacenadas también en el disco, excepto las que hayan sido declaradas como locales mediante la primitiva LOCAL, que solamente existen mientras las usa el procedimiento en el que han sido declaradas (o en sus descendientes).

VALORP

Si al lanzar el programa la lista INGLES no tiene valor asignado, hay que ejecutar INICIALIZA. Para lo cual viene como anillo al dedo la primitiva VALORP, cuya entrada es un nombre de variable y cuya salida es VERDAD, si esta variable tiene asignado un valor, y FALSO, en caso contrario. Según esto, el programa quedaría así:

```
SEA TRADUCTOR.5
SI NO VALORP "INGLES [INICIALIZA]
TRADUCIR
FIN
```

siendo el resto igual que en las versiones 3 ó 4.

Este programa admite múltiples variantes que pueden resultar interesantes:

1. Podrían ofrecer, en algunos casos, distintas posibilidades de traducción. Por ejemplo, al escribir ARE el programa podría contestar con ERES SOMOS SOIS SON. ¿Cómo podrían asignarse varias palabras a una sola? La respuesta está en las listas. Prueba el procedimiento ASIGNA con:

```
HAZ "INGLES [BOOK THE ARE SILLY]
HAZ  "ESPAÑOL  [LIBRO  [EL LA LOS LAS] [ERES SOMOS SOIS SON]
[TONTO TONTA TONTOS TONTAS]]
```

Si vas a hacer asignaciones de este tipo dentro de TRADUCIR, ten cuidado, ya que a PALTRADUCIDA se le asigna el primero de la lista que tecleas. Tendrías que distinguir si la asignación va a ser de una palabra o de una lista.

2. No es excesivamente difícil elaborar un programa que traduzca frases literalmente, incluyendo en él la opción para elegir entre varias acepciones de una palabra. Puedes considerar la frase como una lista de palabras y, para una interpretación literal, basta ir traduciendo palabra por palabra hasta terminar con la lista. Insistiremos en esta cuestión.

3. Se le puede "dar la vuelta" al programa para que en lugar de traducir palabras sirva para hacer ejercicios de vocabulario. El programa escribiría una palabra en inglés y el usuario tendría que contestar con su traducción; el programa indicaría si ésta es correcta o no. Caso de no serlo, puede optarse por pasar a otra palabra o dar una nueva

AZAR

oportunidad al usuario. Para elegir palabras al azar puedes usar la primitiva AZAR, cuya entrada es un número, N, y cuya salida es un número aleatorio entre 0 y N-1. Prueba este procedimiento:

```
SEA ALEA
(TECLEA (1 + AZAR 6) CAR 32)
ALEA
FIN
```

Como AZAR 6 produce números aleatorios entre 0 y 5, al sumarle 1 los tenemos entre 1 y 6, de modo que este procedimiento simula los puntos que pueden salir al tirar un dado.

CUENTA

La primitiva CUENTA nos proporciona el número de elementos de una lista. Su entrada es la lista en cuestión (en el IBM puede ser también una palabra), y su salida es el número de elementos que posee.

Con ella podemos hacer una cómoda elección al azar, ya que

```
CUENTA :INGLES
```

es el número de elementos de la lista INGLES;

```
1 + AZAR CUENTA :INGLES
```

es un lugar cualquiera de esta lista;

```
ITEM (1 + AZAR CUENTA :INGLES) :INGLES
```

es, con toda seguridad, un elemento de la lista.

3.10
Jugando a los dados

La primitiva AZAR permite simular multitud de situaciones y comportamientos, siendo insustituible cuando se trata precisamente de juegos de azar. Seguramente habrás presenciado, en algunas películas americanas, escenas en las que se juega a los dados, apostándose hasta las pestañas. Este juego, *craps*, tiene las siguientes reglas:

Si el que tira saca un 7 o un 11, gana; si saca otro punto, continúa tirando hasta que vuelva a sacar el mismo punto (en cuyo caso gana), o saque 7 u 11 (en cuyo caso pierde).

Estas reglas del juego constituyen casi un programa; lo único que hace falta es repetir unas cuantas partidas para ver cuáles son las posibilidades de ganar del que tira los dados.

El procedimiento DADOS simula el lanzamiento de dos dados devolviendo la suma de sus puntos.

```
SEA DADOS
DEVUELVE SUMA (1 + AZAR 6)(1 + AZAR 6)
FIN
```

Una partida sería:

```
SEA PARTIDA
LOCAL "PUNTOS
HAZ "PUNTOS DADOS
TEST MIEMBROP :PUNTOS [7 11]
SIVERDAD [HAZ "GANADAS :GANADAS + 1]
SIFALSO [SIGUE.PARTIDA]
FIN

SEA SIGUE.PARTIDA
LOCAL "PUNTOS1
HAZ "PUNTOS1 DADOS
TEST :PUNTOS = :PUNTOS1
SIVERDAD [HAZ "GANADAS :GANADAS + 1]
SIFALSO [SI NO MIEMBROP :PUNTOS1 [7 11] [SIGUE.PARTIDA]]
FIN
```

Es evidente que GANADAS hay que ponerlo a 0 al empezar el programa. Si juegas cien partidas, ¿cuántas crees que ganarías? Elabora tu hipótesis y contrástala con el resultado del procedimiento CRAPS

```
SEA CRAPS
HAZ "GANADAS 0
PARTIDAS 100
(ESCRIBE [HAS GANADO] :GANADAS)
FIN

SEA PARTIDAS :NUMERO
SI :NUMERO = 0 [ALTO]
PARTIDA
PARTIDAS :NUMERO - 1
FIN
```

Hay un inconveniente en este programa, que seguramente habrás detectado. Hasta que no termine, y tardará en hacerlo, no sabrás "cómo va". Arréglalo para que al menos diga algo cada vez que ganes; tardará más en la ejecución, pero psicológicamente te parecerá que el tiempo empleado es menor.

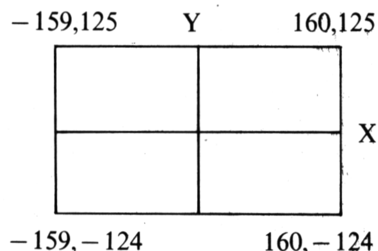
Seguramente conocerás más juegos. Quizá pretendas programar algunos para conocer tus posibilidades de ganar.

Gráficos

En multitud de ocasiones habrás tenido que dibujar curvas. Un procedimiento de "fuerza bruta", pero eficaz en muchos casos, consiste en dibujar "un montón" de puntos de la curva. Cuando usas una

calculadora y papel, "el montón" no suele pasar de 20. No estaría nada mal dibujar algunos más, y el poderoso LOGO también va a ayudarte en esto:

La tortuga puede moverse en un sistema de referencia cartesiano cuyo origen es el centro de la pantalla. El eje X va, en IBM, de −159 a 160 pasos de la tortuga, y el eje Y de −124 a 125. En APPLE estos números son −140, 139, −120 y 119, respectivamente.



PUNTO

Además, la primitiva PUNTO, independiente de la tortuga, dibuja un punto en la pantalla en el lugar indicado por la lista, que es su entrada; el primer elemento de la lista es, como de costumbre, la coordenada X del punto, y el segundo, la coordenada Y.

Escribe en una línea ET, para esconder la tortuga, y lo que sigue:

```
PUNTO [0 0]
PUNTO [5 5]
PUNTO [10 10]
PUNTO [15 15]
PUNTO [20 20]
PUNTO [25 25]
PUNTO [30 30]
```

y observa el efecto que produce.

3.11 Lluvia

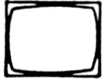
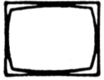
Si quieres simular pequeñas gotas de lluvia en la pantalla, usa el procedimiento

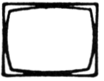
```
SEA LLUVIA
HAZ "X -159 + AZAR 320
HAZ "Y -124 + AZAR 250
PUNTO LISTA :X :Y
LLUVIA
FIN
```

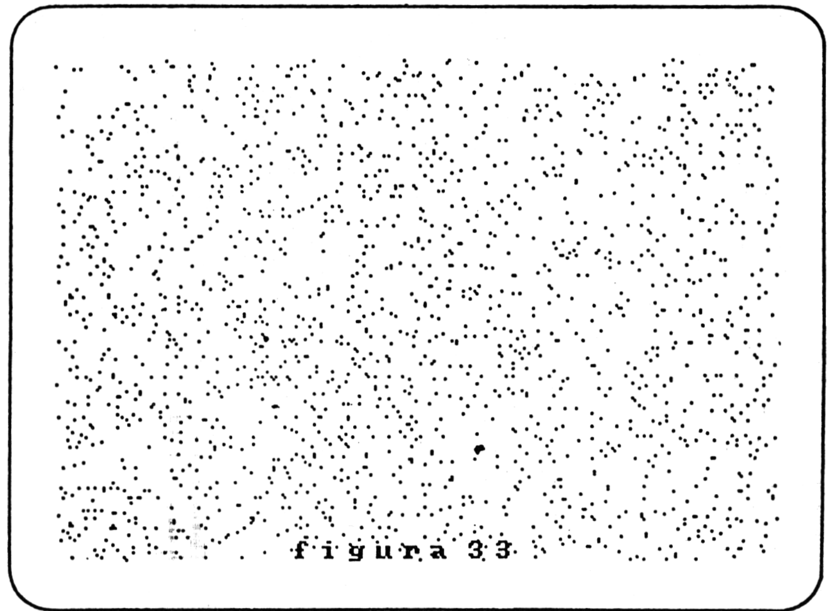
escribiendo ET GRAFICOS LLUVIA, y déjalo funcionar un rato.

LISTA

La primitiva LISTA devuelve la lista formada por sus entradas (para más de dos hay que poner todo entre paréntesis), y la usamos aquí porque la entrada de PUNTO debe ser una lista. ¿Por qué no funciona si escribes PUNTO [:X :Y]?





3.12

Gráfica de una función

Continuemos con lo nuestro. Para dibujar la curva puedes empezar dando a X el valor −159, calcular la Y correspondiente y dibujar el punto (X,Y); posteriormente, aumentar la X en un par de unidades y repetir la operación, hasta llegar al valor 160. Escrito en LOGO sería:

```
SEA DIBUJA :X
SI :X > 160 [ALTO]
HAZ "Y <valor de la función>
PUNTO LISTA :X :Y
DIBUJA :X + 2
FIN
```

Este procedimiento te proporcionará 160 puntos de la función, que no está nada mal, o al menos eso parece. Para terminar el procedimiento basta con aclarar ⟨valor de la función⟩. En principio habría que escribir la expresión de la función. Por ejemplo, si quieres dibujar la recta Y = 0.5 * x + 30, la línea se convertiría en:

```
HAZ "Y 0.5 * :X + 30
```

Entrada de instrucciones por el teclado

Tiene el inconveniente (que en otros lenguajes de programación es inevitable) de que para representar otra función habría que reescribir esta línea del programa.

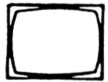
En LOGO podemos hacer algo mucho mejor: que el programa lea siempre la función que debe representar. Podemos escribir, entonces:

EJECUTA

```
HAZ "Y EJECUTA :FUNCION
```

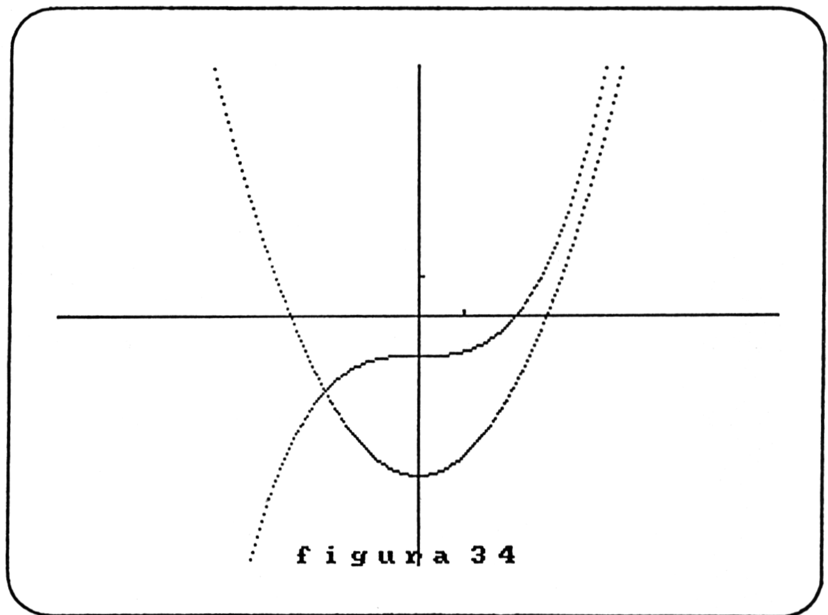
donde FUNCION será la expresión de la función que deberás teclear durante la ejecución del programa CURVAS1

```
CURVAS1
ESCRIBE [ESCRIBE LA EXPRESION DE LA FUNCION]
HAZ "FUNCION LEELISTA
DIBUJA -159
ESCRIBE [TERMINADO]
FIN
```








figura 34

La primitiva EJECUTA tiene como entrada una lista y su efecto es ejecutar ésta como si fuese una instrucción dada desde el nivel superior. Por ejemplo, si el valor de la variable FUNCION es la lista [0.5 * :x + 30] y X vale 10, EJECUTA :FUNCION producirá 35.

Como ves, no tienes nada que modificar. Siempre que ejecutes CURVAS1 el programa te pedirá la expresión de la función; la tecleas y listo.

Por supuesto tienes que escribir en la notación LOGO, es decir, si quieres dibujar $y = 0.005 * x^2 + 10$, tendrás que teclear:

```
0.005 * :x * :x + 10
```

¿Que es incómodo? Puedes evitarlo si utilizas tus conocimientos sobre manejo de listas. Transforma entonces [0.005 * X * X + 10] en su correspondiente expresión LOGO.

Si construyes el procedimiento TRANSFORMA, que devuelva la expresión transformada, CURVAS1 habría que modificarlo así:

```
HAZ "F.LEIDA LEELISTA
HAZ "FUNCION TRANSFORMA :F.LEIDA
```

¿Cuál es el cometido de TRANSFORMA? Tomar cada palabra de F.LEIDA y, si es una X, ponerla en la lista FUNCION como :X, esto es, como la palabra formada por : y X; si no es una X, hay que dejarla como está. A ver qué te parece este procedimiento:

```
SEA TRANSFORMA :F
SI VACIAP :F [DEVUELVE []]
TEST PRIMERO :F = "X
SIVERDAD [DEVUELVE FRASE (PALABRA ": "X) (TRANSFORMA
SINPRIMERO :F)]
SIFALSO [DEVUELVE FRASE (PRIMERO :F) (TRANSFORMA SINPRIMERO
:F)]
FIN
```

FRASE

FRASE es una nueva primitiva que devuelve una lista formada por sus entradas (para más de dos hay que poner paréntesis). ¿Qué diferencia hay entre ésta y LISTA? Compruébalo tú mismo, comparando los resultados de las siguientes instrucciones:

```
ESCRIBE LISTA "UN "DIA
ESCRIBE FRASE "UN "DIA
ESCRIBE LISTA "UN [BUEN DIA]
ESCRIBE FRASE "UN [BUEN DIA]
ESCRIBE (LISTA [HOY ES] "UN [BUEN DIA])
ESCRIBE (FRASE [HOY ES] "UN [BUEN DIA])
ESCRIBE LISTA [HEMOS TERMINADO] []
ESCRIBE FRASE [HEMOS TERMINADO] []
```

Ahora ya sabes de qué va.

Este programa de dibujo de funciones tiene varios inconvenientes:

No dibuja los ejes coordenados.
No admite escalas.

Sólo dibuja en el intervalo (−159, 160).
Si se produce algún error de cálculo, el programa "se rompe".

Prueba a darle como función RAC X y verás lo que ocurre. Esto último podrías arreglarlo, en este caso particular, sustituyendo en CURVAS1, DIBUJA −159 por DIBUJA 0, pero no es una buena solución, ya que tendrías que introducir modificaciones para cada curva particular.

POS

Lo de los ejes tiene fácil arreglo gracias a la tortuga. La primitiva POS, cuya entrada es una lista, mueve a la tortuga desde donde esté hasta la posición indicada por la lista, dejando su trazado habitual si está en activo el lápiz. Entonces

```
SEA EJES
ET
SL POS [-159 0]
CL POS [159 0]
SL POS [0 -124]
CL POS [0 124]
FIN
```

Hemos introducido ET porque a la tortuga no la necesitamos en la gráfica. Incorpora EJES a CURVAS1 y tendrás CURVAS2. Más adelante volveremos sobre los otros problemas, pero tú puedes ir pensando ya en ellos.

VALPOS

Puesto que has utilizado POS, te conviene conocer su complementaria, VALPOS. Esta primitiva no tiene entradas, y su salida es una lista que da la posición de la tortuga en ese momento.

3.13 Control de la pantalla: nieve

Quizá más llamativo que el efecto "lluvia" sea el efecto "nieve", que puede conseguirse escribiendo el símbolo "*" en la pantalla de textos, de la misma forma que escribías PUNTO en la pantalla gráfica.

Vamos a mostrarte algunas primitivas útiles para este propósito.

La pantalla tiene, en LOGO, 25 filas, numeradas de 0 a 24, siendo 0 la fila superior, y 40 columnas, numeradas de 0 a 39, siendo 0 la columna más a la izquierda.

ANCHURA
VALANCHURA
CURSOR

En IBM se puede poner una anchura entre 2 y 80 utilizando la primitiva ANCHURA con un número como entrada; VALANCHURA devuelve la anchura que tenga en ese momento la pantalla.

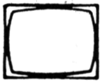
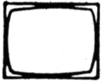
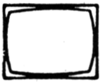
La primitiva CURSOR, cuya entrada es una lista con dos números (fila y columna en IBM, columna y fila en APPLE), coloca el cursor en la posición indicada por esa lista (la columna 39 no es

accesible el cursor, ya que la utiliza el sistema para escribir la marca de continuación de línea). Prueba esto:

```
LT CURSOR [12 19] TECLEA "*
```

Si te damos

```
SEA NIEVA
LT
HAZ "COPOS []
NIEVE
DERRITE :COPOS
FIN
```



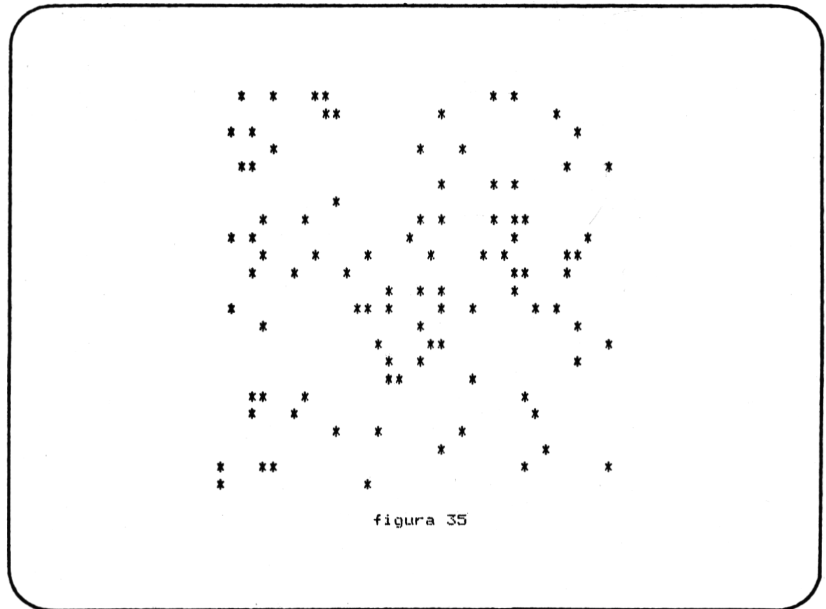


figura 35

es con la sana intención de que escribas un NIEVE análogo a LLUVIA, salvo que en el programa NIEVE se almacenará cada COPO (es decir, su posición) en la variable COPOS, de manera que se pueda utilizar esta lista en DERRITE, procedimiento que borra los copos que han caído. Puedes parar, por ejemplo, cuando hayas puesto cien copos. DERRITE se limitará a escribir un espacio en blanco en cada valor de la lista COPO almacenado en la lista COPOS. Aunque la nevada será más lenta, puedes conseguir que no caigan dos copos en el mismo sitio. ¿Cómo?

3.14

Un problema de mínimos

Vamos a ver cómo puedes utilizar tus conocimientos para resolver un problema. Por ejemplo, deseamos construir una ventana rectangular que tenga una luz de 1 metro cuadrado, en una pared de 3 metros de largo por 2,20 de alto, y queremos gastar el mínimo posible en la madera del marco. ¿Qué dimensiones le daremos a éste?

Veamos. La longitud del marco es:

$$\text{Perímetro} = 2 * \text{base} + 2 * \text{altura} = 2 * (\text{base} + \text{altura})$$

Por otra parte, el área es:

$$\text{Area} = \text{base} * \text{altura} = 1$$

de manera que el perímetro puede expresarse así:

$$\text{Perímetro} = 2 * (\text{base} + 1/\text{base})$$

Podemos dar a la base valores mayores que 0 y menores que 3, y obtener los diferentes valores del perímetro mediante el procedimiento:

```
SEA PERIMETRO :BASE
DEVUELVE 2 * (:BASE + 1 / :BASE)
FIN
```

Por ejemplo, prueba:

```
ESCRIBE PERIMETRO 0.5
ESCRIBE PERIMETRO 1
ESCRIBE PERIMETRO 1.5
ESCRIBE PERIMETRO 2
ESCRIBE PERIMETRO 2.5
```

¿Cuál es el valor mínimo? ¿Estás seguro? Se podría afinar más probando en torno al valor 1 de la base, que parece el candidato más firme; por ejemplo, desde 0,5 hasta 1,5, con incrementos de 0,1.

Claro está que sería mejor diseñar un procedimiento que se encargase de ir dando valores a la base y calcular el valor del perímetro obtenido en cada caso, almacenando unos y otros hasta encontrar el mínimo.

Una posibilidad es empezar almacenando en BASE.MIN 0,1 y en PER.MIN PERIMETRO 0,1. Teniendo estos valores nos disponemos a calcular los de PERIMETRO para 0,2, 0,3, etc., comparando cada vez con el valor de PER.MIN; si encontramos uno menor

lo almacenamos en PER.MIN, y de paso almacenamos también el correspondiente valor de BASE.MIN, continuando en caso contrario hasta haber agotado los valores de BASE que queremos analizar. Algo así:

```
SEA MINIMO :BASE
LOCAL "PER.TEMP
SI :BASE > :BASE.FINAL [ALTO]
HAZ "PER.TEM PERIMETRO :BASE
SI :PER.TEMP <  :PER.MIN  [HAZ  "BASE.MIN :BASE HAZ "PER.MIN
:PER.TEMP]
MINIMO :BASE + 0.1
FIN
```

Una vez efectuado, sólo quedaría por escribir algo como:

```
ESCRIBE [DE LOS VALORES QUE HE CALCULADO]
(ESCRIBE [LA BASE QUE DA EL MINIMO ES] :BASE.MIN)
(ESCRIBE [LA ALTURA QUE DA EL MINIMO ES] 1/:BASE.MIN)
(ESCRIBE [EL PERIMETRO MINIMO ES] :PER.MIN)
```

y, por supuesto, tienes que integrarlo todo en un procedimiento que dé los valores iniciales que ya hemos indicado para BASE.MIN, PER.MIN, BASE.FINAL, llame a MINIMO y escriba los resultados. ¿Podría haberte ayudado GRAFICAS a resolver este problema?

3.15
El cazador y su presa

Vamos a poner de manifiesto nuevas habilidades de la tortuga mediante un juego en el que ella será el cazador y un punto móvil sobre la pantalla será la presa.

Imagina un punto en la pantalla que se mueve de izquierda a derecha sobre la recta $y = 100$, por ejemplo, avanzando cada vez una distancia constante. La tortuga, por su parte, empezará a moverse desde su posición original en (0,0), orientándose hacia la presa y avanzando una distancia también constante. Los dos movimientos podemos integrarlos en un procedimiento que dé valores iniciales y comience la persecución.

```
SEA PERSECUCION
HAZ "X.PRESA -159
HAZ "Y.PRESA 100
HAZ "AVANCE.PRESA 5
HAZ "AVANCE.CAZADOR 5
PERSIGUE
FIN
```

```
SEA PERSIGUE
MOV.PRESA
MOV.CAZADOR
PERSIGUE
FIN
```

Una versión de DIBUJA nos ayudará a trazar el punto que constituye la presa

```
SEA MOV.PRESA
HAZ "X.PRESA SUMA :X.PRESA :AVANCE.PRESA
PUNTO LISTA :X.PRESA :Y.PRESA
FIN
```

Orientación

Vamos a ver ahora el movimiento del cazador, la tortuga. Hemos dicho que ésta se va a orientar hacia su presa y después avanzará una cierta distancia. Esto último no presenta ninguna dificultad; pero ¿cómo dirigir la tortuga hacia el punto-presa? Hay dos primitivas que nos permitirán una fácil solución de este problema: RUMBO y HACIA.

RUMBO

RUMBO tiene como entrada un número de grados, y su efecto es dejar a la tortuga apuntando hacia ese rumbo. Prueba estos valores:

```
RUMBO 0
RUMBO 45
RUMBO 90
RUMBO 180
RUMBO 270
RUMBO 0
RUMBO -90
RUMBO -180
RUMBO -270
RUMBO -315
```

VALRUMBO

Detrás de cada uno de los cuatro últimos usa la primitiva VALRUMBO, que no tiene entradas y cuya salida es un número de grados mayor o igual que 0 y menor que 360. Así podrás observar cómo funciona.

Para proporcionar una entrada a RUMBO habría que ver cuál es el ángulo formado por la dirección "sur-norte" y la dirección cazador-presa. Por supuesto, sabiendo las posiciones de ambas y un poco de trigonometría podrías hacerlo. Pero la primitiva HACIA te ahorra ese trabajo. Su entrada es la lista formada por las coordenadas cartesianas de un punto, y su salida, el rumbo necesario para orientar la tortuga hacia dicho punto. Prueba

HACIA

```
RUMBO HACIA [10 10]
RUMBO HACIA [10 0]
RUMBO HACIA [0 -10]
RUMBO [HACIA [-10 0]
```

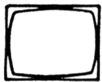
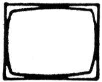
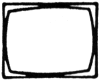
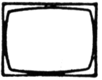
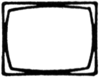
Con estas primitivas el movimiento del cazador es fácil de escribir. Aquí lo tienes:

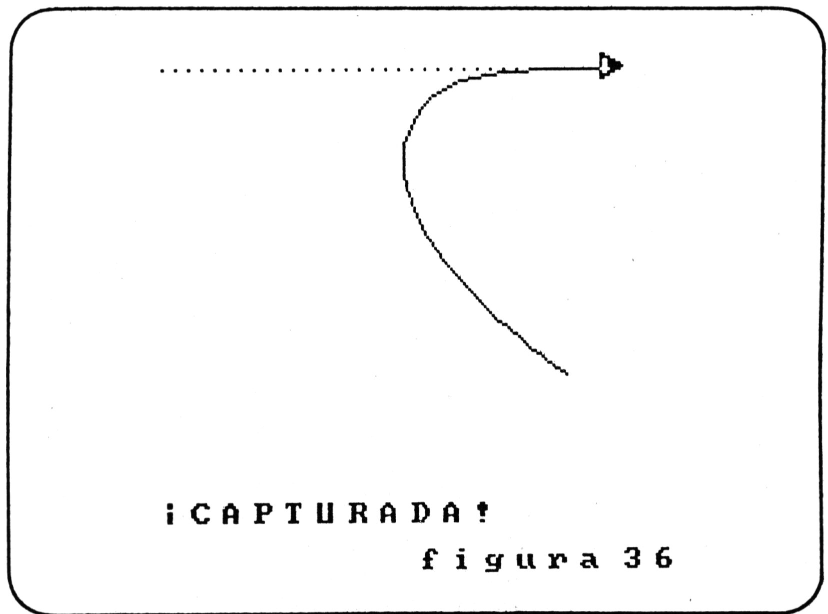
```
SEA MOV.CAZADOR
RUMBO HACIA LISTA :X.PRESA :Y.PRESA
AVANZA :AVANCE.CAZADOR
FIN
```

Este juego te servirá como entretenimiento y hasta para estudiar. Aplica distintos valores iniciales a las variables y observa cómo se comporta. Habrás comprobado, aun antes de jugar, que el juego no se detiene, aunque no estaría de más pararlo cuando el cazador alcance a la presa.

Para ello bastaría con comprobar si la posición de la presa, dada por X.PRESA Y.PRESA, coincide con la de la tortuga, dada por VALPOS, en cuyo caso detendríamos PERSIGUE escribiendo ¡CAPTURADA!

Ensaya esta condición para cerciorarte de que generalmente no funciona. ¿Por qué?



Porque en estos casos deben utilizarse condiciones de distancia, algo así como

```
SI DISTANCIA LISTA :X.PRESA :Y.PRESA VALPOS < :DIST [ESCRIBE
[¡CAPTURADA!] ALTO]
```

en la que DIST es una distancia prefijada, que puede recibir valor junto con las demás variables en PERSECUCION. La distancia entre las posiciones (A,B) y (P,Q) es, como ya dijo Pitágoras, la raíz cuadrada de $(A - P)^2 + (B - Q)^2$.

El procedimiento DISTANCIA podría escribirse así

```
SEA DISTANCIA :L1 :L2
(LOCAL "D1 "D2)
HAZ "D1 DIFERENCIA PRIMERO :L1 PRIMERO :L2
HAZ "D2 DIFERENCIA ULTIMO :L1 ULTIMO :L2
DEVUELVE RAC SUMA :D1 * :D1 :D2 * :D2
FIN
```

Este procedimiento trabajo perfectamente en IBM, ya que en él DIFERENCIA es una primitiva. En APPLE necesitarás el procedimiento DIFERENCIA. Un obsequio:

```
SEA DIFERENCIA :A :B
DEVUELVE SUMA :A PRODUCTO -1 :B
FIN
```

Incorpora al procedimiento la condición de terminación, que para algunos valores de los avances de presa y cazador no va a funcionar, ya que el cazador no alcanzará a la presa.

Por otra parte, en lugar de empezar con el cazador en el centro de la pantalla, podrías situarlo en cualquier otra posición, por ejemplo en la parte inferior de la misma. ¿Qué tendrías que hacer para comenzar así?

¿Podrías mover a la presa sobre la recta $y = 50$? ¿Podrías desplazarla sobre la recta $y = x$, empezando el cazador en la esquina inferior derecha de la pantalla? ¿Podrías moverla sobre una circunferencia de radio 100 alrededor del cazador, situado en el centro?

Conduciendo desde el teclado

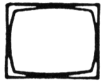
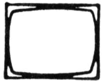
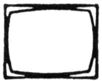
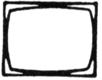
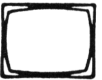
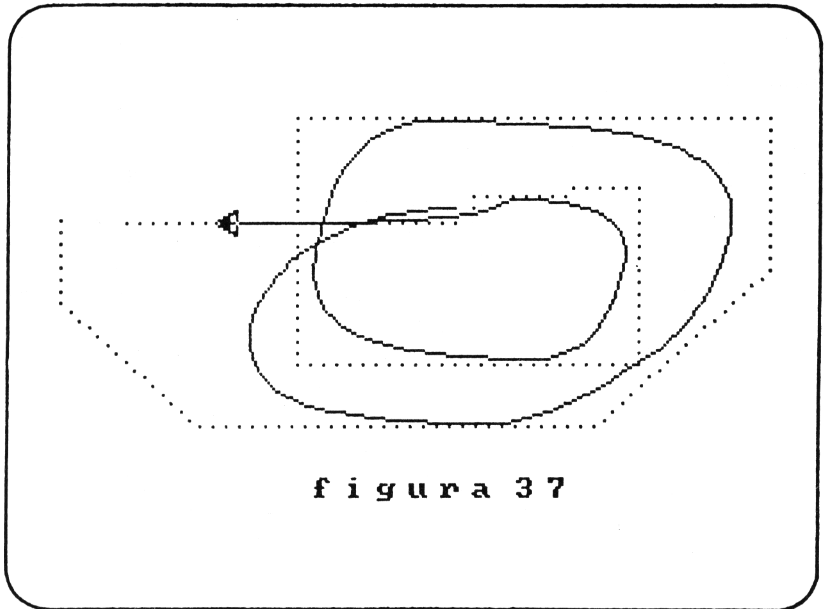
Una posibilidad interesante es que tú hagas de presa. No te alarmes, lo que queremos decir es que la controles desde el teclado, utilizando, por ejemplo, cuatro teclas que dirijan el movimiento hacia arriba, abajo, izquierda y derecha. La primitiva LEECAR, que ya has usado, permite realizar esta operación con facilidad.

Si queremos que la presa se desplace hacia arriba, basta con dejar su X como está y aumentar su Y; si deseamos un movimiento hacia la derecha, aumentaremos su X y dejaremos su Y como está, etc.

En IBM puedes utilizar el teclado numérico, que se activa pulsando la tecla marcada NUM LOCK. En APPLE puedes usar cuatro

teclas que estén en cruz. El único procedimiento que hay que modificar es MOV.PRESA, que quedaría así (para IBM):

```
SEA MOV.PRESA
LOCAL "MOV
HAZ "MOV LEECAR
SI :MOV = "8 [HAZ "Y.PRESA SUMA :Y.PRESA :AVANCE.PRESA]
SI :MOV = "2 [HAZ "Y.PRESA DIFERENCIA :Y.PRESA
:AVANCE.PRESA]
SI :MOV = "6 [HAZ "X.PRESA SUMA :X.PRESA :AVANCE.PRESA]
SI :MOV = "4 [HAZ "X.PRESA DIFERENCIA :X.PRESA
:AVANCE.PRESA]
PUNTO LISTA :X.PRESA :Y.PRESA
FIN
```







figura 37

Con este procedimiento puedes montarte PERSECUCION1. Pruébalo y a ver qué tal te portas como presa. Por cierto, ¿qué ocurre si no pulsas ninguna tecla? ¿Te conviene eso o prefieres otra alternativa?

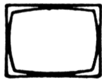
Aun a riesgo de que pienses que estamos incordiándote, vamos a decirte que podrías mejorar el movimiento de la presa. ¿Sería factible disponer de ocho direcciones en lugar de cuatro? Usando el teclado numérico podrían tenerse en cuenta también las teclas 1, 3, 7 y 9.

Un pequeño problema: si deseamos que el avance de la presa sea siempre la longitud AVANCE.PRESA, ¿cuánto habrá que aumentar la X y la Y al pulsar la tecla 9?

Las relaciones trigonométricas nos dan la solución :AVANCE-PRESA * COS 45 y :AVANCE.PRESA * SEN 45 para X e Y, respectivamente.







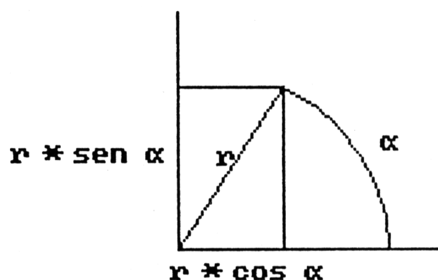


figura 38

Como los ángulos correspondientes a las otras teclas son 135, 225 y 315, no hay ningún problema en reescribir un nuevo procedimiento para el movimiento de la presa.

Quizá fuera mejor utilizar una lista, POS.PRESA, que almacenara la posición de la presa.

En PERSECUCION tendrías que suprimir X.PRESA e Y.PRESA, así como sus inicializaciones, sustituyéndolas por

```
HAZ "POS.PRESA [-159 100]
```

y también LISTA :X.PRESA :Y.PRESA por POS.PRESA donde proceda.

En este PERSECUCION2 el procedimiento del movimiento de la presa tendría instrucciones tales como

```
SI :MOV = "8 [HAZ "POS.PRESA SUMALIST :POS.PRESA LISTA 0
:AVANCE.PRESA]
SI :MOV = "3 [HAZ "POS.PRESA SUMALIST :POS.PRESA LISTA
:AVANCE.PRESA * COS 135 :AVANCE.PRESA * SEN 135]
```

y por supuesto

```
PUNTO :POS.PRESA
```

El procedimiento que recibe dos listas y devuelve la lista suma de ambas término a término es sencillo.

```
SEA SUMALIST :L1 :L2
DEVUELVE LISTA (SUMA PRIMERO  :L1  PRIMERO :L2) (SUMA ULTIMO
:L1 ULTIMO :L2)
FIN
```

¿Cómo te queda PERSECUCION2?

3.16 Conduciendo la tortuga

Hemos utilizado procedimientos que permiten gobernar un punto desde el teclado. ¿Se puede hacer algo parecido con la tortuga? Desde luego. Empecemos con algo sencillo

```
SEA CONDUCE1
LOCAL "MOV
HAZ "MOV LEECAR
SI :MOV = "D [DERECHA 10]
SI :MOV = "I [IZQUIERDA 10]
AVANZA 5
CONDUCE1
FIN
```

Pruébalo conduciendo un tiempo la tortuga. Observarás que, mientras no pulses una tecla, la tortuga permanecerá quieta, y eso no es conducir.

Es mejor construir un procedimiento al que podemos llamar LEETECLA, en lugar de LEECAR

```
SEA LEETECLA
SI TECLAP [DEVUELVE LEECAR]
DEVUELVE "
FIN
```

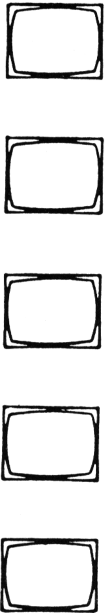
TECLAP

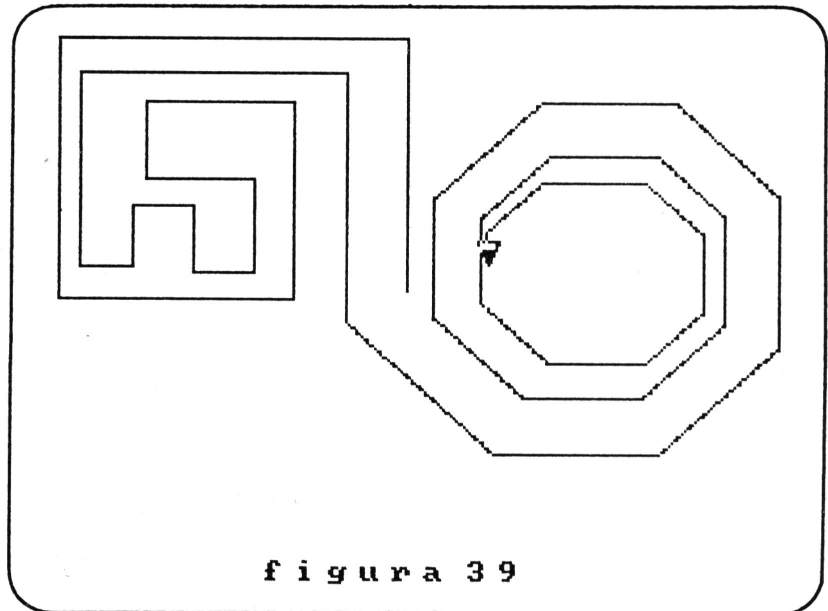
La primitiva TECLAP no tiene entradas y devuelve VERDAD o FALSO según se haya pulsado o no alguna tecla. Tal como está el procedimiento, si se pulsa una tecla devuelve el carácter correspondiente, y si no es así, devuelve la palabra vacía, lo cual produce AVANCE 5 en CONDUCE1. Reescribe CONDUCE1 de esta forma y ensáyalo.

Puedes utilizar también ocho direcciones haciendo que la tortuga se desplace una cierta distancia según cada una de ellas, pero ahora es más sencillo, ya que la tortuga dispone de RUMBO.

CONDUCE2 podría ser algo así

```
SEA CONDUCE2
LOCAL "MOV
HAZ "MOV LEETECLA
SI :MOV = "8 [RUMBO 0]
SI :MOV = "9 [RUMBO 45]
SI :MOV = "6 [RUMBO 90]
.....................
AVANZA 5
CONDUCE2
FIN
```





Los puntos suspensivos, a tu cargo. Prueba este procedimiento para hacer dibujos en la pantalla.

Copiando dibujos en impresora

Si un dibujo te agrada quizá desees guardar una copia en papel. En IBM basta con pulsar la tecla marcada con PrtSc *(Print Screen)*; ten presente que este rótulo está en la parte superior de la tecla, de modo que debes pulsar al mismo tiempo la de mayúsculas.

Si lo haces así, todo el contenido de la pantalla pasa a la impresora, sea texto o gráficos; por tanto, si prefieres que sólo aparezca el dibujo es conveniente que escribas GRAFICOS antes de pulsar la tecla.

En APPLE resulta algo más complicado, pero a continuación te facilitamos un procedimiento que permite enviar un dibujo a la impresora y ponerle un título.

.IMPRESORA

```
SEA IMPRIME.PANT
LOCAL "TITULO
(TECLEA [¿TITULO?] CAR 32)
HAZ "TITULO LEELISTA
.IMPRESORA 1
ESCRIBE CAR 17
ESCRIBE :TITULO
.IMPRESORA 0
FIN
```

Escribiendo en la impresora

.IMPRESORA 1 permite que a partir de ese momento todo lo que se escriba vaya a parar a la impresora, de modo que la escritura del carácter 17, no imprimible, pone en marcha el proceso de copia de la pantalla gráfica. Puesto que el flujo de información está "desviado" hacia la impresora, la escritura del título se producirá en ésta en lugar de en la pantalla. A partir de .IMPRESORA 0 la información volverá a aparecer en la pantalla. Puede conseguirse que la información aparezca en ambos lugares escribiendo

```
.IMPRESORA 9
```

pero no olvides que continúa siendo necesario que escribas .IMPRESORA 0 cuando no quieras que las instrucciones aparezcan en la impresora.

En IBM el tratamiento es ligeramente distinto, ya que su LOGO considera a la impresora como un archivo, y antes de escribir en él hay que abrirlo. Para eso se utiliza la instrucción

ABRE

```
ABRE "LPT1
```

donde LPT1 *(line printer 1)* es el nombre que el sistema da a su impresora. Una vez hecho esto, desde el momento en que se teclee

ESCRIBIR

```
ESCRIBIR "LPT1
```

los resultados de ejecutar ESCRIBE, TECLEA y MUESTRA van a parar a la impresora en lugar de a la pantalla. Esto sucederá hasta que teclees

`ESCRIBIR "CON` (consola)

y desde ese momento aparecerán en la pantalla todos los "mensajes".

MUESTRA

Por cierto, MUESTRA es una nueva primitiva, análoga a ESCRIBE. Para que observes su funcionamiento, escribe

```
MUESTRA "HOLA
MUESTRA [HOLA]
```

Listando programas en la impresora

¿Cómo procederás si quieres imprimir un programa? Existen multitud de opciones. Si lo que deseas es imprimir algo que tienes en la memoria central, basta con que escribas

```
GUARDA "LPT1
```

y ya está. También puedes poner

```
GUARDA "LPT1 "<paquete>
```

donde ⟨paquete⟩ es el nombre de un paquete que está en la memoria central.

Asimismo puedes conseguir que la impresora se comporte como un espía que copia todo lo que aparece en pantalla. Si escribes

ESPIA

```
ESPIA "LPT1
```

a partir de ese momento cualquier cosa que pase en la pantalla quedará también en la impresora. Por ejemplo, ECPS permite que todos los procedimientos que haya en la memoria se escriban en la pantalla, y también en la impresora. Análogamente ocurriría con ECTS, etc.

Si deseas imprimir un archivo del que en este momento no dispongas en la memoria central, basta con que escribas

ECARCHIVO

```
ECARCHIVO "CIRCUNS.LF
```

para que pase a la pantalla y a la impresora. Cuando quieras acabar con la condición de espía de la impresora escribe

NOESPIA

```
NOESPIA
```

Guardando dibujos en el disco

En IBM tienes también la posibilidad de guardar en el disco un dibujo que esté en la pantalla gráfica. Si, por ejemplo, utilizando

CONDUCE2 has dibujado una cara y quieres guardarla en el disco directamente, basta con que escribas

GUARDADIB

```
GUARDADIB "CARA
```

Te encontrarás entonces con que en el directorio del disco aparece el archivo CARA.PIC. Este archivo puedes traerlo a la pantalla utilizando

TRAEDIB

```
TRAEDIB "CARA
```

Esta posibilidad de guardar varios dibujos y pasarlos después a la pantalla puede servir para realizar algún dibujo animado modesto.

A estas alturas ya debes manejarte con cierta soltura en LOGO. Por tanto, vamos a dedicar el capítulo siguiente, MISCELANEA, a una selección de programas, algunos de los cuales son ampliación de los que ya hemos visto.

ANTE
NTE
ANTECEDEN
E LEELISTA
NTE = []
CHAZ"ANTE
: ANTECED
AND LEE

4
Miscelánea

Este capítulo es una colección de programas, algunos de cierta complejidad, cuyo desarrollo te proporcionará un considerable dominio del lenguaje LOGO y, en determinados casos, un primer contacto con algunos problemas interesantes de dominios como el de la "inteligencia artificial".

Naturalmente, cuanto mayor es el interés de un programa, mayores dificultades suele presentar, tanto en su concepción como en su desarrollo. Anímate y adelante.

Los programas que vamos a desarrollar son:

1. *Adivina un número*

Describe un juego en el que se trata de adivinar un número elegido al azar por el programa. Se plantea el problema del número necesario de intentos para acertar un número entre 1 y N. Finalmente se completa el programa de modo que haga trampas.

2. *La tortuga dinámica*

Amplía los programas CONDUCE y permite crear un modelo de tortuga que se mueve "sin rozamiento", haciendo intervenir velocidades y aceleraciones. Se indica cómo realizar un juego para conducir esta tortuga por una pista.

3. *La tortuga dibujante*

Indica cómo utilizar la tortuga para trazar dibujos en la pantalla de modo que se puedan corregir y almacenar posteriormente.

4. *Algo más sobre gráficas*
Termina con los problemas que se plantearon anteriormente en los procedimientos CURVAS.

5. *Construcción aleatoria de frases*
Comienza con un programa que construye frases al azar a partir de sujetos, verbos y predicados, que se le proporcionan, completándose después de modo que escriba todas las frases posibles con los datos de que dispone.

6. *Más sobre traducción*
Se encarga del problema de la traducción literal de frases, con las opciones que ya se indicaron al discutir TRADUCTOR.

7. *¿Declinar? ¡Pero si es muy fácil!*
Indica cómo diseñar un programa que permita desarrollar cualquiera de las declinaciones latinas.

8. *¿Necesitas un psiquiatra?*
Versión muy simplificada del famoso ELIZA de Weizenbaum, que permite hacerse una idea del funcionamiento de este tipo de programas.

9. *Animal: árboles de información*
Explica cómo utilizar una estructura arbórea para almacenar información creando un programa que aprende.

10. *El Nim*
Se desarrolla el famoso juego según una idea de Papert y Solomon, presentando los tres niveles de tanteador, árbitro y jugador que puede tener el programa.

11. *Instrucciones de programación estructurada*
Para los amantes de la misma, se explica cómo diseñar en LOGO todas las instrucciones de control.

12. *El juego de la vida*
La evolución de una población de "células" a lo largo del tiempo, vista en la pantalla del ordenador, y unas cuantas ideas para otros juegos de este tipo.

13. *Cuerpos celestes*
La aplicación de las leyes de la Mecánica de Newton permite simular en pantalla los movimientos de estrellas dobles, satélites, etc.

14. *Curva sorprendente*
Se indica cómo dibujar una curva recursiva con una sorprendente propiedad, que puede ser estudiada utilizando el programa que la dibuja.

15. *Resolución de ecuaciones*
Utilizando el método del punto medio se estudia un programa

para resolver ecuaciones, de toda laya y condición, con una sola incógnita.

16. *Tres dimensiones*

Se cuenta cómo construir un programa que represente superficies en tres dimensiones, lo que produce en muchos casos efectos verdaderamente impresionantes.

17. *Traza*

Este programa encierra una cierta dificultad, ya que se trata de modificar otros programas para que al ejecutarlos se clarifique su funcionamiento, "trazando" el valor de las variables en cada momento, indicando el procedimiento en curso, etc., de forma que puedan detectarse errores de funcionamiento.

18. *Experto*

Es, quizá, la estrella de todos ellos. Indica cómo construir un programa que saca consecuencias lógicas a partir de una base de hechos y unas reglas, sirviendo para resolver problemas de lógica simbólica. Permite también la explotación de una pequeña base de datos mediante relaciones definidas en ella.

Y esto es todo, aunque podría ser más. En algún lugar hay que pararse. De cualquier manera, estos programas ofrecen ideas suficientes para animarte a utilizar LOGO, su forma de trabajar en la resolución de tus propios problemas.

4.1
Adivina un número

Es fácil hacer un programa donde el usuario tiene que adivinar un número elegido por el propio programa. Por ejemplo,

```
SEA ADIVINA
HAZ "NUM.ELEGIDO 1+AZAR 100
INTENTALO
FIN

SEA INTENTALO
(TECLEA [¿NUMERO?] CAR 32)
SI :NUM.ELEGIDO = LEENUMERO [ESCRIBE "ACERTASTE ALTO]
INTENTALO
FIN
```

LEENUMERO es el procedimiento que ya conoces para filtrar entradas numéricas.

Así las cosas, está claro que el número acaba adivinándose, pero ¿y si fijamos el número de intentos de que dispone el jugador? Supón que incluimos en ADIVINA

```
HAZ "TOTAL.INTENTOS 6
```

Para contar el número de intentos podrían utilizarse la cabecera y primera línea siguientes:

```
SEA INTENTALO :INTENTOS
SI :INTENTOS > :TOTAL.INTENTOS [FALLASTE]
```

donde FALLASTE es el procedimiento que informa al usuario de que han terminado sus posibilidades de adivinar el número y le da su más sentido pésame. Todavía habría que modificar algo más, así que hazlo ahora mismo para ahorrarte problemas.

En estas condiciones es prácticamente imposible adivinar el número; inténtalo y verás. Hagamos el programa "más amigable", de modo que indique al usuario por dónde van los tiros. Supón que funcionara así:

```
¿NUMERO? 50
EL MIO ES MAS ALTO
¿NUMERO? 75
EL MIO ES MAS ALTO
¿NUMERO? 88
EL MIO ES MAS BAJO
```

etcétera. En este caso las posibilidades del jugador aumentan notablemente con la utilización de la búsqueda binaria (¿?) sugerida por el programa. Para incluirlo en el programa podríamos utilizar una variable local, NUMERO, en la que se almacenaría el número tecleado por el usuario; comparado el valor de NUMERO con el de NUM.ELEGIDO, habría que escribir los mensajes oportunos, deteniendo el juego si se ha acertado y felicitando al jugador, por supuesto.

Quizá merezca la pena, y esto es frecuente cuando se hace un programa, volver sobre INTENTALO antes de continuar, insertando línea tras línea en él. Podría reescribirse así:

```
SEA INTENTALO :INTENTOS
TEST :INTENTOS > :TOTAL.INTENTOS
SIVERDAD [FALLASTE]
SIFALSO [SI NO.ACERTADO [INTENTALO INTENTOS + 1]
FIN
```

NO.ACERTADO sería el procedimiento que se encargase de leer el número, comparar, etc., y devolver VERDAD O FALSO, según el caso. ¿Qué te parece éste?

```
SEA NO.ACERTADO
LOCAL "NUMERO
(TECLEA [¿NUMERO?] CAR 32)
HAZ "NUMERO LEENUMERO
TEST :NUM.ELEGIDO = :NUMERO
SIVERDAD [ACERTASTE :INTENTOS DEVUELVE "VERDAD]
SIFALSO [SI :NUM.ELEGIDO > :NUMERO [ESCRIBE [EL MIO  ES  MAS
ALTO]] [ESCRIBE [EL MIO ES MAS BAJO]]]
SIFALSO [DEVUELVE "FALSO]
FIN
```

ACERTASTE debe dar la enhorabuena indicando el número de intentos empleados para acertar el número. En FALLASTE se debería decir al jugador del número elegido por el programa, ya que aunque las máquinas no hacen trampas (¿?) hay mucha gente desconfiada por ahí.

Es conveniente que, una vez concluida una partida, se pueda continuar jugando o no. Esto requeriría algunas líneas más del tipo que ya conoces:

```
ESCRIBE [¿OTRA PARTIDA? (S/N)]
<jugar otra vez si se teclea S>
```

¿Dónde vas a ponerlas? Antes de decidirlo piensa que si se juegan varias partidas se podrían dar "puntos", empezando con

```
HAZ "PUNTOS 500
```

encargándose ACERTASTE y FALLASTE de aumentar y quitar 100, respectivamente; también habría que comunicarle al jugador los puntos que tiene. ¿Sería aconsejable colocarlas antes de preguntar por otra partida?

Si juegas un poco con el programa verás que el 6, como número de intentos, es razonable. ¡En 7 intentos se puede acertar siempre! Cambia TOTAL.INTENTOS a 7 y asegúrate que es verdad.

A continuación te planteamos un par de problemas para que perfecciones el juego. En lugar de limitarse a los números entre 1 y 100, podría pensarse en los números entre 1 y NUMERO.FINAL, siendo éste un número elegido por el jugador, de manera que el programa se comportase así:

```
ELIGE UN NUMERO ENTRE 1 Y 1024
800
ELEGIRE UN NUMERO ENTRE 1 Y 800
TE DOY 9 INTENTOS PARA ADIVINARLO
```

y después continuaría como antes.

Tu problema consiste en averiguar cuál debe ser el valor que debe

tener TOTAL.INTENTOS conociendo NUMERO.FINAL. Experimentar con lápiz y papel, o la práctica con el programa, te permitirán ver que si NUMERO.FINAL está entre 4 y 7 puede adivinarse con seguridad en tres intentos; si está entre 8 y 15, puede adivinarse con seguridad en cuatro intentos; si está entre 16 y 31, puede hacerse en cinco. Y no te decimos más. Busca el número de intentos que permiten acertar con seguridad para un NUMERO.FINAL cualquiera; el número de intentos razonables, TOTAL.INTENTOS, puedes tomarlo como una unidad menor. ¡Ah!, no admitas números menores de 4, porque entonces resulta muy sencillo. Incorpora tus descubrimientos al programa y pruébalo.

El otro problema que queremos plantearte está relacionado con algo que hemos dicho antes: una máquina no puede hacer trampas. ¿Qué quiere decirse cuando se hace una afirmación así? Desde luego, un programador sí puede y, si esto no hiere tu sensibilidad, esperamos que completes este juego añadiendo un procedimiento que haga trampas.

La idea es la siguiente: si el número de puntos acumulado por el jugador es, por ejemplo, 800, en lugar de pasar a ADIVINA nuevamente, se pasa a TRAMPAS. Este procedimiento se puede construir en forma análoga a como se construyó ADIVINA, y debe dar exactamente los mismos mensajes, aunque:

> No necesita molestarse en elegir un número para compararlo.
>
> El procedimiento INTENTALO podrías utilizarlo, sustituyendo NO.ACERTADO por NO.ACERTARAS (esto supone que tienes que reescribirlo con otro nombre; quizá sirva NI.LO.INTENTES).
>
> NO.ACERTARAS es la parte más delicada. Tiene que dar los mismos mensajes que NO.ACERTADO, pero el problema es saber cuándo hay que escribir si el número es alto o es bajo, ya que no tienes con quién comparar. ¿Qué te parece decirlo al azar? ¿Le ves algún problema? Sea como sea, no olvides que al final tendrás que decir cuál era el número "elegido" por el programa.

Una sugerencia: el primer número que va a decir el jugador es 50, casi seguro. Si el programa contesta que el suyo es más alto, deberá elegir el siguiente entre 51 y 100, lo que supone 50 números, y si dice que es más bajo, deberá elegir entre 1 y 49, 49 números. El artero y malévolo NO.ACERTARAS debería entonces dirigir al jugador hacia el primero. Generalmente, cada vez que el jugador teclee un número el procedimiento deberá dirigirlo hacia el intervalo donde haya más números.

Queda pendiente el problema de dar el número "elegido". Apáñatelas. ¿Te ayudaría saber en qué intervalo se está eligiendo en cada momento?

Si practicas con el programa comprobarás que se puede elegir un número y además hacer trampas. ¿Cómo?

La tortuga dinámica

Un clásico en LOGO es la tortuga dinámica. Consiste en crear una tortuga que se mueve por la pantalla en forma análoga a como lo hace una nave por el espacio, sin rozamiento. Antes de empezar con ella, y para que veas las diferencias, te sugerimos que desempolves el programa CONDUCE y hagas lo siguiente:

Dibuja dos circunferencias concéntricas, con centro en el origen, de radios 60 y 80, por ejemplo. Coloca la tortuga en (70,0) y ponla en marcha. ¿Cuántas vueltas eres capaz de mantenerte en la pista? No estaría nada mal que el procedimiento se detuviese cuando esto ocurra; existe un procedimiento, DISTANCIA, que puede ayudarte a resolver el problema.

Otra cuestión: ¿Puedes contar el número de vueltas que da la tortuga a la pista? La respuesta debe ser afirmativa. En caso contrario, revisa el apartado de los POLIGONOS.

Si pruebas con distintos valores de giros y avances en combinación con los radios de las circunferencias, podrás ofrecer distintos niveles de dificultad en el juego.

Y vamos con la tortuga dinámica (¡perdón!, el navío espacial). Dispone de un cohete que le permite cambiar la velocidad, es decir, acelerar en la dirección que apunta el morro del cohete. Cuenta también con pequeños cohetes laterales que le permiten cambiar su rumbo, pero no su velocidad. Veamos cómo se puede mover nuestro navío. Si en un determinado momento su velocidad viene dada por el vector (3,1) y en ese instante está en la posición (2,7), en el instante siguiente se encontrará en la posición (5,8), en el siguiente en la (8,9), en el siguiente en la (11,10), etc., de acuerdo con la interpretación de la velocidad como variación de la posición por unidad de tiempo.

Puesto que la primitiva VALPOS da la posición de la tortuga y VELOCIDAD puede almacenar su velocidad en un momento dado, colocar la tortuga en la nueva posición será simplemente

```
POS SUMA.LIST :VALPOS :VELOCIDAD
```

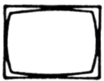
siendo SUMA.LIST un procedimiento que devuelve la lista formada por la suma de los elementos de sus entradas.

De manera que el procedimiento que mueve la tortuga podría ser

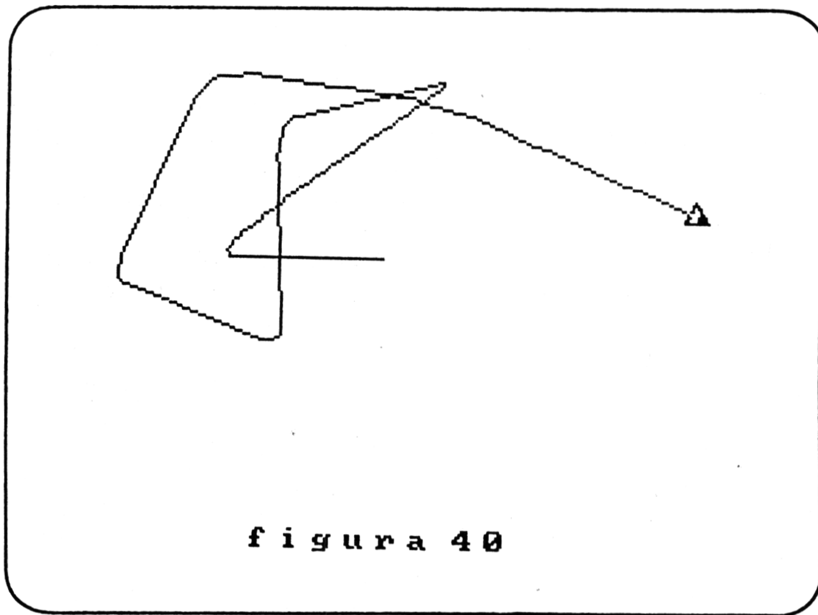
```
TORTUDINA
LOCAL "MOV
HAZ "MOV LEETECLA
SI :MOV="D [DERECHA 30]
SI :MOV="I [IZQUIERDA 30]
SI :MOV="A [ACELERA]
```

```
POS SUMA.LIST :VALPOS :VELOCIDAD
TORTUDINA
FIN
```








figura 40

Hará falta fijar la velocidad inicial, pero ésta, junto con la posición inicial, podemos asignarla en un procedimiento, DINA, que llame a TORTUDINA.

ACELERA es el procedimiento que se va a encargar de darle un nuevo valor a VELOCIDAD, sumándole la aceleración proporcionada por el "disparo del cohete". Puesto que la aceleración se produce en la dirección que tenga el navío en ese momento, comprueba que si el rumbo de la tortuga es el ángulo A y consideramos el módulo de la aceleración como 1, las componentes de la misma son (sen A, cos A).

Puesto que nuestro ángulo A no es otra cosa que VALRUMBO, y la nueva velocidad es la anterior más la aceleración, el procedimiento sería simplemente

```
SEA ACELERA
HAZ "VELOCIDAD SUMA.LIST :VELOCIDAD LISTA SEN VALRUMBO COS
VALRUMBO
FIN
```

Inicializa en DINA la velocidad con el valor [0 0]. Con esta velocidad inicial, si no pulsas ninguna tecla, la tortuga permanecerá en la misma posición. Si pulsas I o D, cambiará de rumbo, pero se

quedará donde está. Cuando pulses A empezará el baile. Te sugerimos lo siguiente:

> Ejecuta DINA pulsando A y déjala correr.
> Coloca la tortuga con rumbo 90, ejecuta DINA, pulsa A y déjala correr.
> Maniobra como quieras, pero ten presente que tal como va el navío, si te limitas a variar el rumbo, "irá de lado", ya que el cambio no se produce hasta que aceleres. Familiarízate con ella, porque es mucho más difícil de conducir que la otra. Llegar a cosmonauta es difícil.

En lugar de utilizar una aceleración de módulo 1 puedes emplear una de módulo MOD.A, que puedes inicializar en DINA, escribiendo:

```
HAZ "MOD.A 1
```

Tendrás que modificar ACELERA, ya que ahora la aceleración sería la lista:

```
:MOD.A*SEN VALRUMBO :MOD.A*COS VALRUMBO
```

Hemos puesto una aceleración variable para permitirte modificarla en TORTUDINA, aumentándola o disminuyéndola. Como las letras A y D están ocupadas, puedes utilizar S y B (subir o bajar), escribiendo

```
SI :MOV="S [HAZ "MOD.A :MOD.A + 1]
SI :MOD="B [HAZ "MOD.A :MOD.A - 1]
```

Efectúa estas modificaciones en DINA y TORTUDINA; si no tocas S ni B, estarás en las mismas condiciones que en el primer proyecto. Si empleas un IBM, quizá prefieras usar el teclado numérico, que resultará más cómodo para el piloto.

Un buen comandante de navío espacial debería conocer su rumbo y posición en cada momento. Podrías utilizar las líneas 23 y 24 de la pantalla para que se viera esta información; recuerda la primitiva CURSOR.

Puesto que ya has conducido una tortuga por una pista, y has tenido que integrar CONDUCE dentro del juego, ¿por qué no haces lo mismo para la tortuga dinámica?

Cuando hayas terminado con ello puedes intentar otra actividad. Dibuja una pequeña circunferencia, de radio 20, por ejemplo, en una esquina de la pantalla y conduce tu nave hasta su interior.

Seguramente se te ocurrirá algo más. Ponlo en práctica.

4.3

La tortuga dibujante

Ya has visto cómo utilizar la tortuga para hacer dibujos en la pantalla, y se supone que tendrás un procedimiento que te permita tenerla perfectamente controlada. De todos modos es probable que, aun así, te equivoques. Entonces tienes que borrar todo y volver a empezar, lo que no garantiza en absoluto que no te vuelvas a equivocar. Veamos cómo solventar este problema usando la primitiva EJECUTA:

Cuando pulses una tecla, la orden correspondiente, además de ejecutarse, se almacenará en una lista, LISTA.DIB. Si se produce un error se presionará una tecla, que puede ser C, para activar un procedimiento que elimina de la lista la orden correspondiente a la última tecla de dibujo pulsada, borra la pantalla y vuelve a dibujar, de modo que el error ya no estará en el dibujo y puedes continuar tranquilamente. Empecemos por el principio:

```
SEA TORTUDIB
HAZ "LISTADIB []
DIBUJA
FIN

SEA DIBUJA
LOCAL "MOV
HAZ "MOV LEECAR
SI :MOV="A [EJECUTA.ALMACENA [AVANZA 5]]
SI :MOV="R [EJECUTA.ALMACENA [RETROCEDE 5]]
SI :MOV="I [EJECUTA.ALMACENA [IZQUIERDA 15]]
SI :MOV="D [EJECUTA.ALMACENA [DERECHA 15]]
SI :MOV="C [CORRIGE.DIBUJA]
SI :MOV="F [ALTO]
DIBUJA
FIN
```

Observarás que usamos LEECAR en lugar de LEETECLA para poder dibujar tranquilamente, así como la inclusión de un retroceso para facilitar los dibujos.

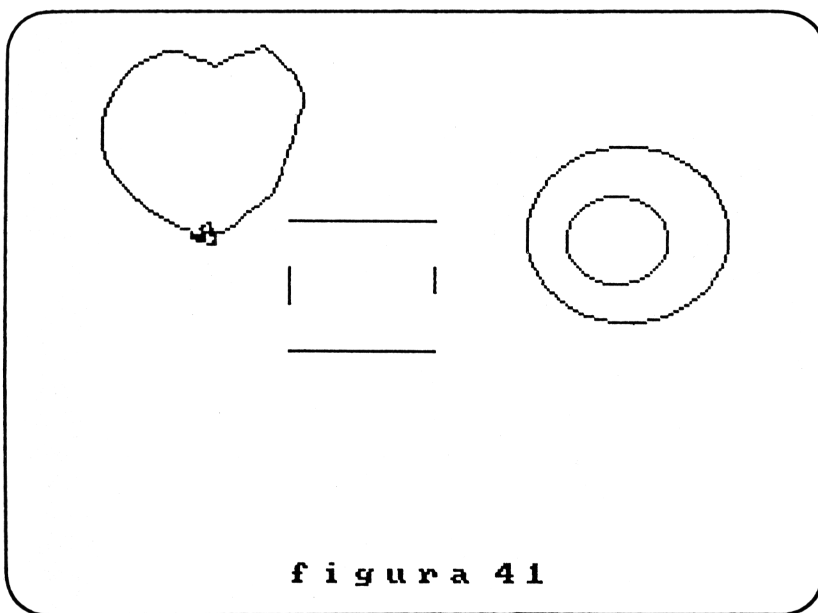
EJECUTA.ALMACENA no debe tener problemas:

```
SEA EJECUTA.ALMACENA :ORDEN
EJECUTA :ORDEN
<colocar la orden en el último lugar de LISTADIB>
FIN
```

CORRIGE.DIBUJA tampoco debe dar demasiado trabajo

```
SEA CORRIGE.DIBUJA
<suprimir el último elemento de LISTADIB>
LG
DIBUJAR :LISTADIB
FIN
```





La misión de DIBUJAR es, simplemente, ir ejecutando cada una de las órdenes que hay en LISTADIB; la primera, la primera de la lista sin la primera..., ya sabes.

Antes de continuar debemos advertirte que, tal y como está CORRIGE.DIBUJA, puede presentar problemas en un caso extremo. Si lees atentamente lo que se ha escrito sabrás a qué nos referimos.

Si quieres puedes aumentar las posibilidades de DIBUJA, añadiendo, por ejemplo: con lápiz, sin lápiz, etc.

Pruébalo para dibujar un mapa o cualquier otra figura que desees. Si la figura te gusta, ya sabes que en IBM puedes almacenarla en disco sin más que utilizar GUARDADIB.

En el APPLE podrías emplear una solución de emergencia: almacenar la variable LISTADIB, ella sola, en un archivo y asegurarte, cuando guardes TORTUDIB, de que sólo conservas los procedimientos, sin las variables que puedan haberse producido al ejecutarlo. Posteriormente, al traer LISTADIB y TORTUDIB, ejecutando DIBUJAR con LISTADIB como entrada tendrías de nuevo tu dibujo.

Sería muy conveniente poder agrupar todas esas instrucciones bajo un nombre de procedimiento, que pudiera almacenarse y ejecutarse como todos los demás. Increíble pero cierto, ¡puede hacerse!

DEFINE

La primitiva DEFINE tiene dos entradas, un nombre y una lista de listas; su salida es un procedimiento, cuyo nombre es el indicado, dispuesto para su ejecución. Un ejemplo: si escribes

```
DEFINE "POLI [[LADO ANGULO] [AVANZA :LADO] [DERECHA :ANGULO]
[POLI  :LADO :ANGULO]]
```

obtienes el mismo efecto que si escribieras en el editor

```
SEA POLI :LADO :ANGULO
AVANZA :LADO
DERECHA :ANGULO
POLI :LADO :ANGULO
FIN
```

Pruébalo y verás cómo después del DEFINE puedes ejecutar POLI tranquilamente.

Observa que en la lista de listas que define POLI, la primera de ellas está formada por los nombres de las variables que figuran en la cabecera del procedimiento; si el procedimiento que vas a definir no tiene variables debe colocarse una lista vacía. Las demás listas corresponden, cada una, a una línea del procedimiento.

En estas condiciones, si en la variable NOMBRE está almacenado el nombre que se quiere dar al procedimiento, la instrucción

```
DEFINE :NOMBRE PONPRIMERO [] :LISTADIB
```

nos lo deja listo para su uso. Desde luego debería estar en un procedimiento, que puedes llamar PROCEDIMIENTO, que pregunte por el nombre y, una vez definido, llame a TORTUDIB. De este modo podría incorporarse, con la letra P como comando, a las posibilidades de DIBUJA, escribiendo en él

```
SI :MOV="P [PROCEDIMIENTO]
```

Un lujo a tu alcance es incorporar otro procedimiento a la lista de habilidades de DIBUJA, que te permita dar un nombre de procedimiento y conseguir el dibujo correspondiente. Para ello puedes crear un procedimiento HAZ. DIBUJO que podría activarse con el comando H.

DEFINIDOP

El problema puede surgir si te equivocas y pides un procedimiento que no existe: el programa se romperá y perderás lo que tengas en LISTADIB. La solución está en la primitiva DEFINIDOP, cuya entrada es un nombre de procedimiento, y cuya salida es VERDAD, si el nombre corresponde a un procedimiento definido, o FALSO, en caso contrario. Si el nombre del procedimiento está almacenado en la variable NOMBRE, la instrucción que evitaría la rotura, y lo ejecutaría, caso de estar definido, sería:

```
SI NO DEFINIDOP :NOMBRE [ESCRIBE [ESE PROCEDIMIENTO NO ESTA
DEFINIDO]] [EJECUTA :NOMBRE]
```

¿Te animas a escribirlo? Incorpóralo y pruébalo.

Es el momento de decir que si se te ocurre alguna otra cosa útil que quieras incorporar, adelante con ella. A nosotros se nos ocurre una. A ver qué te parece.

Como para hacer un dibujo irás tanteando con longitudes y ángulos, es posible que tu lista sea algo semejante a

```
[AVANZA 5] [AVANZA 5]  [AVANZA  5] [DERECHA 15] [DERECHA 15]
[RETROCEDE 5] [RETROCEDE 5]...
```

A la hora de definir un procedimiento sería mejor dejar esta lista así:

```
[AVANZA 15] [DERECHA 10] [RETROCEDE 10]...
```

agrupando las órdenes consecutivas que tengan el mismo nombre. Suponemos que no vas a tener en la lista instrucciones como [AVANZA 5] [RETROCEDED 5], ya que se trataría de un error y lo habrías corregido.

Para simplificar la lista, una posibilidad, si no se te ocurre otra mejor, es utilizar tres variables: ANTERIOR, ACTUAL y LDIB. Se almacena en ANTERIOR el primero de LISTADIB, se pone LDIB en blanco y se procesa el resto de LISTADIB de la siguiente manera:

El primero del resto se almacena en ACTUAL y se compara ANTERIOR con ACTUAL; si son iguales, por ejemplo, dos AVANZA, se almacena en ANTERIOR un AVANZA 30, y si son distintos se almacena ANTERIOR como el último de LDIB y se pasa el valor de ACTUAL a ANTERIOR, continuando así hasta terminar la lista. Después se devuelve LDIB. Aquí tienes una parte ya hecha:

```
SEA SIMPLIFICA.LISTA
(LOCAL "ANTERIOR "ACTUAL "LDIB)
HAZ "ANTERIOR PRIMERO :LISTADIB
HAZ "LDIB []
SIMPLIFICA SINPRIMERO :LISTADIB
DEVUELVE :LDIB
FIN
```

de este modo, la incorporación de SIMPLIFICA.LISTA al procedimiento es muy simple; basta con escribir:

```
DEFINE :NOMBRE PONPRIMERO [] SIMPLIFICA.LISTA
```

Cuando uses el programa te darás cuenta que LISTADIB crece enseguida desmesuradamente, en cuanto el dibujo sea complejo, de manera que no sería mala idea simplificarla poniendo:

```
HAZ "LISTADIB SIMPLIFICA.LISTA
```

SIMPLIFICA queda a tu cargo.

Y no decimos nada más; claro está que si se te ocurre alguna cosa...

4.4

Algo más sobre gráficas

Vamos a volver ahora sobre el programa que dibuja gráficas de funciones.

Algunas de las objeciones que hicimos al programa CURVAS fueron que:

- No admitía diferentes escalas.
- No tenía corrección de errores.
- Sólo dibujaba en el intervalo (−159,160).

Vamos a empezar a subsanarlos. Por lo que a escalas se refiere, el principio es fácil: el procedimiento FIJA.ESCALA puede preguntar al usuario cuántas unidades de la tortuga equivalen a una de las suyas y recoger la respuesta en la variable ESC. ¿Qué hacemos con ella? Si, por ejemplo, el valor de ESC es 10, eso significa que

10 unidades de la tortuga = 1 unidad del usuario

Como en el eje X las unidades de la tortuga van de −159 a 160, eso significa que las del usuario van a ir de −15.9 a 16.0. Puesto que el procedimiento DIBUJA está "montado" sobre unidades de tortuga, tendrás que modificarlo para que traslade unas a otras. Si usas como cabecera:

```
SEA DIBUJA :XTOR
```

habrá que pasar estas unidades a las del usuario, escribiendo:

```
HAZ "X :XTOR/:ESC
```

para calcular el valor de Y correspondiente a esta X. El cálculo se realiza igual que antes, pero para dibujar hay que pasar esta Y a la de la tortuga:

```
HAZ "YTOR :Y*:ESC
```

y sólo queda por dibujar el punto correspondiente en las coordenadas de la tortuga y llamar a DIBUJA con el valor de XTOR incrementado.

Una escala de este tipo, mayor que 1, "comprime" el intervalo en el que se dibuja.

Si el usuario contestase con 0.1, entonces

0.1 unidades de la tortuga = 1 unidad del usuario

o, lo que es lo mismo,

1 unidad de la tortuga = 10 unidades de usuario

Como las unidades de la tortuga van de -159 a 160 sobre el eje X, las del usuario van entonces de -1590 a 1600. Continuando como antes, comprueba que son válidas exactamente las mismas instrucciones que hemos escrito. Una escala de este tipo "amplía" el intervalo en el que se dibuja.

El nuevo procedimiento DIBUJA permite otras prestaciones. Por ejemplo, tanto X^2 como X^4 y X^6 pasan por los puntos $(-1,1)$ y $(1,1)$, pero si las dibujas con CURVAS, no vas a ver qué diferencia presentan en el intervalo $[-1,1]$ del eje X.

Puesto que la Y de la tortuga va de -124 a 125, podrías usar el nuevo procedimiento dando a la escala un valor de 120, con lo que 120 unidades de la tortuga equivaldrían a una de las tuyas, así que el intervalo de la función sería prácticamente el $[-1.3,1.3]$.

Hay un problema que seguramente no se te habrá escapado; los profesores de matemáticas suelen insistir mucho en él. Convendría marcar las unidades en los ejes. Puede hacerse cuando haya terminado el dibujo: en lugar de escribir TERMINADO simplemente, puedes poner en su lugar un procedimiento que ponga las marcas. Por ejemplo, si el valor de ESC es 10, puedes poner marcas en los ejes en los puntos de la tortuga (10,0) y (0,10), escribiendo:

```
PUNTO LISTA :ESC 1
PUNTO LISTA 1 :ESC
```

El 1 es para que se vea mejor la marca. En este caso la marca significa una unidad de las del usuario, así que cuando :ESC es mayor que 1, convendría escribir el mensaje

```
LAS MARCAS CORRESPONDEN A UNA DE TUS UNIDADES
```

Si :ESC es 0.1, está claro que PUNTO LISTA :ESC 1 es inútil, ya que se va a confundir con el eje; es fácil poner

```
PUNTO LISTA 1/:ESC 1
PUNTO LISTA 1 1/:ESC
```

que en este caso serían 10 unidades de la tortuga, o 100 de las del usuario; pero el mensaje que debe aparecer entonces es

```
LAS MARCAS CORRESPONDEN A 100 DE TUS UNIDADES
```

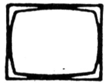
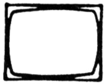
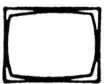
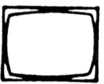
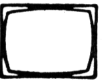
El procedimiento TERMINAR sería, en esquema,

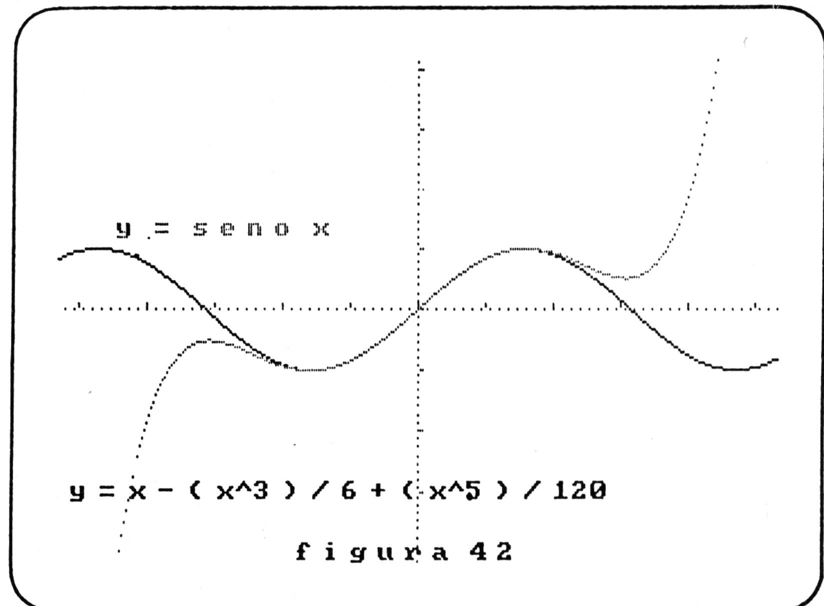
```
SEA TERMINAR
TEST :ESC > 1
SIVERDAD <poner las marcas y escribir el mensaje>
SIFALSO <poner las marcas y escribir el mensaje; el número
de unidades a escribir en el mensaje parece ser
(1/:ESC)*(1/:ESC)>
ESCRIBE [TERMINADO]
FIN
```

Hasta el momento, el programa que podríamos llamar GRAFICAS1 debería tener como procedimientos:

- LEEFUNCION, que se encarga de leer la expresión de la función y traducirla a notación LOGO.
- FIJA.ESCALA, que preguntaría y obtendría el valor de la misma.
- EJES, que se encarga del dibujo de éstos.
- DIBUJA, que traza la curva teniendo en cuenta la escala.
- TERMINAR, que pondría las marcas e indicaría las escalas.

Quizá desees superponer diversas gráficas de funciones distintas, usando la misma escala y sin borrar la pantalla; o dibujar una misma



función con varias escalas distintas para ver cuál es "la que mejor le va". Tú verás cómo tienes que modificar TERMINAR para conseguir esto.

COGE ERROR

Aún persiste el problema del tratamiento de errores. La primitiva COGE y la variable reservada ERROR permiten resolver el problema. Su forma de utilización es la siguiente:

En DIBUJA, escribe las primeras líneas como están y, al llegar a la línea de EJECUTA, escribe:

```
COGE "ERROR [DIB]
SI NO VACIAP ERROR [(ESCRIBE [NO DEFINIDA PARA X=] :X)]
DIBUJA :XTOR+2
```

donde DIB es el procedimiento:

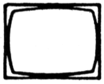
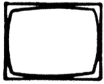
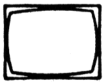
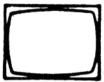
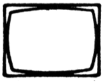
```
SEA DIB
HAZ "Y EJECUTA :FUNCION
HAZ "YTOR :Y * :ESC
PUNTO LISTA :XTOR :YTOR
FIN
```

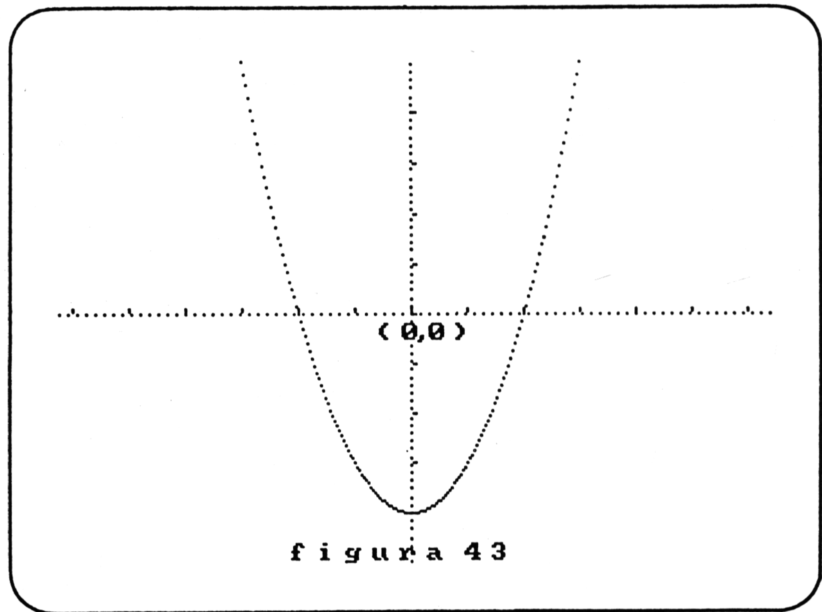
Veamos cómo funciona esto. Cuando se llega a COGE "ERROR se ejecuta la lista de procedimientos que haya a continuación. Si se produce un error de algún tipo en la realización de los procedimientos que hay en esta lista (en este caso, sólo DIB), el control se devuelve automáticamente a la instrucción siguiente a COGE. Esta instrucción dice que si la variable ERROR no es vacía (si ha habido error) se escriba el mensaje indicado, ya que habrá habido problemas con la determinación de Y, tales como raíz de un número negativo, división por cero, etc. En cualquier caso la función no está definida para ese valor de X. Es el usuario, con sus conocimientos de matemáticas, quien debe ver qué es lo que ocurre en cada caso.

Incorpora esta nueva posibilidad a GRAFICAS1 y guárdalo con cuidado; es ya un programa muy completo, que te hubiera permitido tener una idea de dónde estaba el mínimo de la ventana de 1 metro cuadrado de luz. De hecho, este programa puede servirte para resolver ecuaciones, ya que las soluciones de la ecuación $f(x) = 0$ son los puntos de intersección de $y = f(x)$ con el eje X.

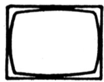
Para que sea realmente eficaz en este último aspecto, deberías poder considerar a la pantalla como una "ventana" que puede moverse a tu voluntad. La idea es la siguiente:

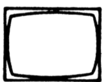
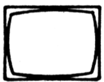
Si dibujas la función $Y = x^2 - 4$, obtendrás lo que ves en la figura 43, en la que el centro de la "ventana" —la pantalla— coincide con el origen de coordenadas. Si "mueves la ventana" de modo que el centro de la misma sea el punto (2,0), deberías obtener algo semejante a lo que observas en la figura 44, en donde las rectas horizontal y vertical no son los ejes coordenados, sino paralelas a ellos, que pasan por el centro de la "ventana".

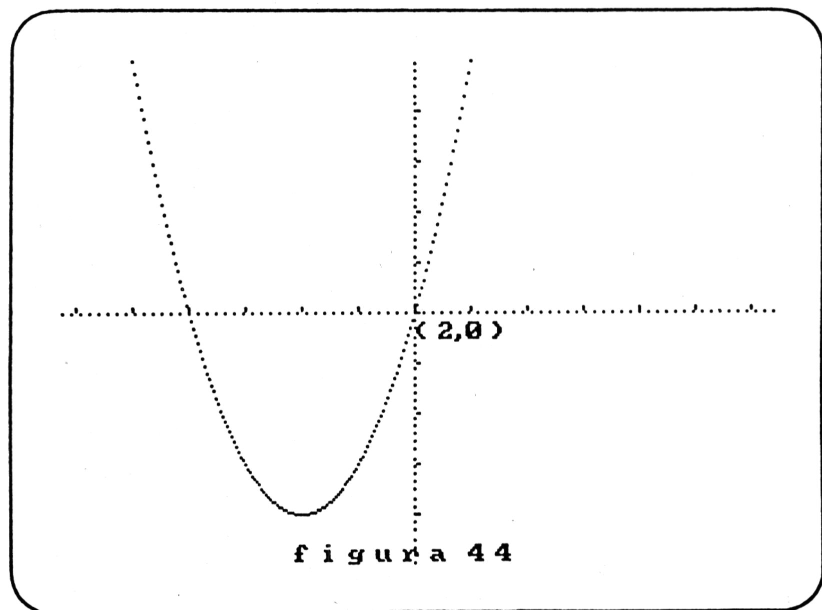








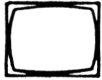
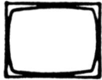
Si observas la figura verás que puedes obtenerla dibujando, con el programa que ya tienes, la función $y = (x + 2)^2 - 4$. Tu problema es entonces saber qué función hay que dibujar para obtener el efecto deseado, partiendo de la función que estás estudiando y conociendo el centro de la "ventana".



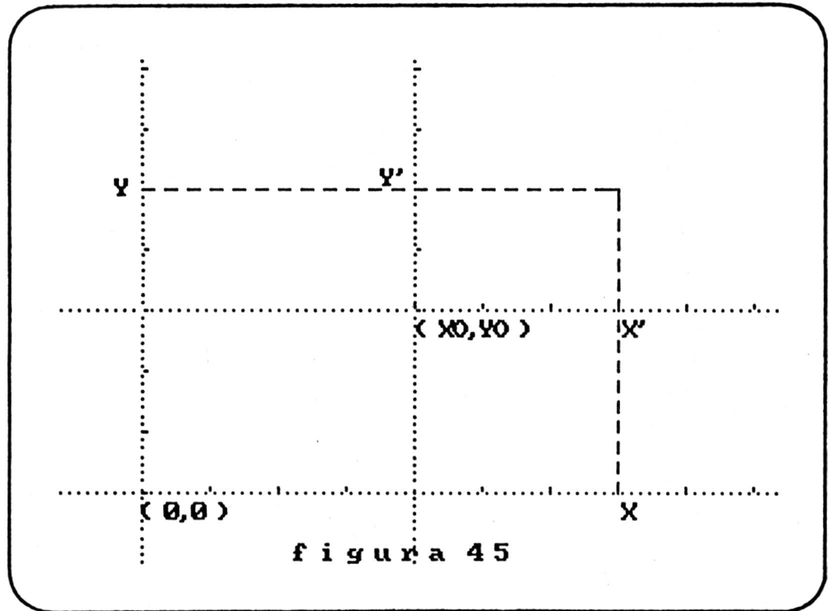






La solución está en considerar el asunto como si, dejando la gráfica en su sitio, trasladases los ejes de modo que el nuevo origen fuera el centro de la "ventana", y calculases la ecuación de la gráfica en este nuevo sistema.



Si te fijas en la figura donde hemos llamado (X,Y) a las coordenadas de un punto en el sistema de origen (0,0) y (X',Y') a las coordenadas del mismo punto en el sistema de referencia cuyo origen es (XO,YO), observarás que se tiene:

```
X = X' + XO
Y = Y' + YO
```

Comprueba que si (XO,YO) es (2,0), al sustituir estas relaciones en $y = X^2 - 4$ se obtiene $y' = (x' + 2)^2 - 4$, de acuerdo con lo que habíamos dicho.

Es evidente que no se trata de hacer las sustituciones "a mano". El programa, después de preguntar por la ecuación de la función, puede hacerlo por las coordenadas del centro de la "ventana", de modo que pueda hacerse la sustitución al traducir a la notación LOGO. De esta manera sólo habría que modificar una línea del procedimiento TRANSFORMA, la correspondiente a SIVERDAD en el TEST sobre PRIMERO :F, que ahora debería ser:

```
SIVERDAD [DEVUELVE FRASE (FRASE ":X "+ ":XO) TRANSFORMA
SINPRIMERO :F]
```

En cuanto a la sustitución de la Y, debe hacerse en el procedimiento DIBUJA. La línea que calcula el valor de Y debería ser ahora:

```
HAZ "Y :YO + EJECUTA :FUNCION
```

Incorpora esta posibilidad al actual programa de gráficas, para obtener uno nuevo, GRAFIV —V por ventana—. No estaría de más que quedase constancia de cuáles son las coordenadas del centro de la "ventana", una vez dibujada la curva. TERMINAR podría dar el oportuno mensaje. En IBM puedes utilizar las posibilidades de "transformista" de la tortuga, así como la posibilidad de dejar "huellas" de su paso, para escribir sobre el dibujo dichas coordenadas.

FORMA

La primitiva FORMA tiene una entrada que puede ser un número entre 0 y 255, o bien "TA. Su efecto es cambiar la imagen de la tortuga a la forma del carácter cuyo ASCII es el número indicado, o bien a la forma familiar si su entrada es "TA.

HUELLA

La primitiva HUELLA carece de entradas. El efecto que produce es dejar en la pantalla gráfica una imagen de la forma actual de la tortuga.

Si, tras almacenar las coordenadas del centro de la "ventana", escribes:

```
HAZ "CENTROV (PALABRA "( :XO ", :YO "))
```

podrás escribir en TERMINAR la línea

```
ROTULO :CENTROV
```

que te escribirá en el dibujo las coordenadas del centro de la ventana. Te obsequiamos con un procedimiento ROTULO en la seguridad de que, una vez lo hayas estudiado, podrás efectuar las modificaciones oportunas para que los rótulos aparezcan donde desees:

```
SEA ROTULO :CENV
SL POS [5 5] RUMBO 90 MT
ROTULO1 :CENV
CENTRO CL FORMA "TA ET
FIN

SEA ROTULO1 :CEN
SI VACIAP :CEN [ALTO]
FORMA ASCII PRIMERO :CEN
HUELLA AV 8
ROTULO1 SINPRIMERO :CEN
FIN
```

Si usas GRAFIV podrás opinar sobre las soluciones de la ecuación $\sqrt{x} = 0.5 * x - 5$.

De cualquier forma, nos ocuparemos en otro lugar de cómo resolver, aproximadamente, esta ecuación y otras más "raras".

4.5

Construcción aleatoria de frases

En algunas ocasiones los profesores se encuentran con que les faltan ejemplos sencillos de frases, en materias tales como Lengua, Idiomas o Latín. Para que puedas echarles una mano vamos a indicarte cómo puedes diseñar un programa que construya frases, eligiendo al azar un sujeto, un verbo y un complemento.

Si inicializas de la siguiente forma

```
HAZ "SUJETOS [[EL PERRO] [EL NIÑO] [JUANA]]
HAZ "VERBOS [[CORRE] [JUEGA] [SALTA]]
HAZ "COMPLEMENTOS [[POR LA CALLE] [INCANSABLEMENTE]  [EN  EL
CAMPO]]
```

la elección de un elemento al azar de una cualquiera de las listas es algo que ya has hecho. Llamando ELIGE al procedimiento que devuelve un elemento de una lista, escribir una frase sería simplemente:

```
ESCRIBE   (FRASE   ELIGE   :SUJETOS   ELIGE  :VERBOS  :ELIGE
:COMPLEMENTOS)
```

tras esta instrucción convendría una del tipo

```
SI TECLAP [ALTO]
```

para que pudieras parar la escritura de frases cuando lo desees.

Llamando FRASE.AZAR al procedimiento que hace esta escritura podrías poner

```
SEA FRASE.AZAR
INICIALIZA
ESCRIBE.FRASE
FIN
```

Ya hemos visto lo que debe hacer INICIALIZA. ¿Es necesario decir que ESCRIBE.FRASE debe llamarse a sí mismo?

En el ejemplo expuesto observarás que si pones en marcha el programa éste empezará a repetir al escribir más de veintisiete frases, o quizá antes.

También hay una cuestión clara: debe conocerse la materia de la que se habla para construir frases con sentido.

Construye tú otros programas que escriban frases no sólo en castellano, sino en otras lenguas. No tienes por qué limitarte a sujeto, verbo y predicado; puedes incluir más elementos.

Tampoco sería mala idea lograr que el programa escriba TODAS las frases posibles utilizando los datos que se le proporcionan. Este problema es equivalente a escribir con las cifras 1 a 3 —sólo había tres sujetos, tres verbos y tres predicados— todos los números posibles de tres cifras. Si diseñas un procedimiento que te escriba:

```
111
112
113
121
122
123
131
132
133
211
212
```

etcétera, entonces no tendrás problemas para escribir las frases, ya que cada cifra puede ser un ITEM de una lista.

Claro está que no es necesario usar todas las listas con el mismo número de elementos. Supongamos que los totales de elementos de cada lista son TOTAL.S, TOTAL.V, TOTAL.C. Estos números son inmediatos de obtener, ya que una vez inicializadas las listas basta con

```
HAZ "TOTAL.S CUENTA :SUJETOS
```

y, asimismo, para los otros totales.

Para producir la anterior serie de números podríamos utilizar el procedimiento cuya cabecera fuese

```
ESCRI.NUM :NS :NV :NC
```

al que llamaríamos escribiendo ESCRI.NUM. 1 1 1.

Lo que debe hacer este procedimiento es funcionar como un "contador":

Se escribe la "frase" :NS :NV :NV y va aumentándose NC.

Cuando el valor de NC pase del TOTAL.C, se da el valor 1 a NC y se aumenta NV en una unidad.

Cuando el valor de NV pase del TOTAL.V se da el valor 1 a NV y se aumenta NS en una unidad.

Cuando NS pase del TOTAL.S nos detenemos.

Comprueba que si desde el nivel superior das valores a TOTAL.S, TOTAL.V y TOTAL.C, y ejecutas el procedimiento

```
SEA ESCRI.NUM :NS :NV :NC
SI :NC > TOTAL.C [HAZ "NC 1 HAZ "NV :NV+1]
SI :NV > TOTAL.V [HAZ "NV 1 HAZ "NS :NS+1]
SI :NS > TOTAL.S [ALTO]
ESCRIBE (FRASE :NS :NV :NC)
ESCRI.NUM :NS :NV :NC+1
FIN
```

obtienes la serie de números indicada.

Comprobado este aspecto, nos interesa el procedimiento ESCRIBE.FRASE. Para ello nos basta con cambiar, en el procedimiento anterior, ESCRI.NUM por ESCRIBE.FRASE, y la línea que produce la escritura por

```
ESCRIBE (FRASE ITEM :NS :SUJETOS ITEM :NV :VERBOS :ITEM  :NC
:COMPLEMENTOS)
```

para obtener un procedimiento que escribe todas las frases.

Si te animas a escribir frases "más ricas", tendrás que modificar el procedimiento ESCRIBE.FRASES de manera que pueda utilizar más variables; pero, a estas alturas, no debe presentar dificultades para ti.

4.6 Más sobre traducción

Ya has visto cómo construir un programa que sirva como un diccionario que va aprendiendo nuevas palabras. Indicamos entonces que se podía intentar realizar una traducción literal de frases. Este nuevo programa utilizaría el INICIALIZA de TRADUCTOR, pero con instrucciones propias, ya que ahora el funcionamiento podría ser algo así:

```
¿FRASE?
THE BOY IS TALL
EL CHICO ES ALTO
¿FRASE?
THE BOY IS UGLY
UGLY NO LA CONOZCO. ¿QUE SIGNIFICA?
FEO
EL CHICO ES FEO
¿FRASE?
ADIOS
?
```

TRADUCIR podrías cambiarlo por TRADUCE.FRASE; es decir:

```
SEA TRADUCE.FRASE
ESCRIBE [¿FRASE?]
HAZ "FRASE.ORIG LEELISTA
SI :FRASE.ORIG = [ADIOS] [ALTO]
TRADUCE :FRASE.ORIG
ESCRIBE :FRASE.TRAD
TRADUCE.FRASE
FIN
```

¿Cómo se traduce una frase? Se toma la primera palabra y, si está en el diccionario, se pone su significado en FRASE.TRAD; si no está,

se pregunta por su significado y, una vez conocido, hay que ampliar el diccionario y poner la acepción en FRASE.TRAD. El procedimiento continúa hasta agotar la lista FRASE.ORIG

```
SEA TRADUCE :FR
(LOCAL "PAL "PAL.TRAD)
SI VACIAP :FR [ALTO]
HAZ "PAL PRIMERO :FR
TEST MIEMBROP :PAL :INGLES
SIVERDAD [PONULTIMO VALOR :PAL :FRASE.TRAD]
SIFALSO [PREGUNTA :PAL PONULTIMO :PAL.TRAD :FRASE.TRAD]
TRADUCE SINPRIMERO :FR
FIN
```

El procedimiento PREGUNTA, que sirve para ampliar el diccionario, no debe presentar problemas con lo que ya sabes.

Como una palabra puede tener varios significados, podría escribirse otra versión que permitiera al usuario elegir entre ellos a la hora de traducir. Recuerda que eso ocurría cuando VALOR :PAL era una lista; si repasas lo hecho en TRADUCTOR, puedes escribir esta versión.

Evidentemente este traductor es muy pobre. Consuélate pensando que el problema de la traducción está todavía sin resolver, aunque se han conseguido progresos notables en dominios restringidos del lenguaje. Para traducir bien una frase hay que comprender su significado, saber qué ha querido decir el que la dice; y éste es un problema difícil. ¿Estás de acuerdo?

En los primeros tiempos del problema de la traducción inglés-ruso y viceversa, para probar los programas se les alimentaba con una frase en inglés que se traducía al ruso; la frase en ruso se suministraba nuevamente al programa para que la tradujera al inglés, comparándose entonces las dos frases inglesas para observar sus discrepancias. Se cuenta que cierto programa recibió la frase

El espíritu es fuerte, pero la carne es débil

y devolvió como retraducción

El vodka es bueno, pero la carne está podrida.

En una novela de ciencia-ficción, funcionarios jubilados con acceso al ordenador de su antiguo lugar de trabajo entretienen sus ocios jugando con "antiguos" programas de traducción. El jugador introduce una frase y entrega a su oponente la retraducción que le ha devuelto la máquina; éste debe adivinar cuál era la frase original. Te ponemos una sencilla, con pista y todo: "gran olla de la nave" es un autor del Barroco español.

¿Declinar? ¡Pero si es muy fácil!

Para declinar hace falta un poco de memoria, cosa que le sobra a tu ordenador, y algunos conocimientos de latín, que se supone son tu aportación personal.

Podrías empezar por algo así:

```
SEA DECLINA
INICIALIZA
DECLINAR
FIN
```

El procedimiento INICIALIZA debería almacenar cuanto hay que recordar para poder declinar: qué es lo que identifica a cada declinación, los casos que escribiremos de ellas y sus terminaciones. Algo así:

```
SEA INICIALIZA
HAZ "PRIMERA "AE
HAZ "SEGUNDA "I
HAZ "TERCERA "IS
HAZ "CUARTA "US
HAZ "QUINTA "EI
HAZ "CASOS [ACS GEN DAT ABL]
HAZ "TERMI.PRI [AM AE AE A]
HAZ "TERMI.SEG [UM I O O]
HAZ "TERMI.TER [EM IS I E]
HAZ "TERMI.CUA [UM US UI U]
HAZ "TERMI.QUI [EM EI EI E]
FIN
```

Para DECLINAR necesitas lo siguiente:

Conocer el nominativo y genitivo de la palabra, ya que con este último puedes saber cuál es la raíz de la palabra y cuál su terminación.

Usando la terminación identificas la declinación, con lo cual sabes qué terminaciones hay que añadir a la raíz para escribir los distintos casos.

Ponerte a declinar.

Según esto, DECLINAR podría ser

```
SEA DECLINAR
PREGUNTA
IDENTIFICA
DECLINANDO
FIN
```

Es evidente que PREGUNTA se va a encargar de preguntar por el nominativo y el genitivo, almacenarlos y lanzar BUSCA, usando

como entrada el genitivo, para determinar provisionalmente la raíz. Puesto que cuatro declinaciones están identificadas por terminaciones de dos letras, AE, IS, US, EI, podemos empezar almacenando en TERMINACION las dos últimas letras del genitivo, que es la palabra de entrada al procedimiento; por tanto, en principio tendríamos:

```
SEA BUSCA :GEN
HAZ "TERMINACION PALABRA ULTIMO SINULTIMO :GEN ULTIMO :GEN
```

Decide qué debes almacenar en RAIZ.

No es por fastidiar, pero observa que, si bien cualquier palabra de la quinta declinación tiene un genitivo en EI, no todos los genitivos en EI son de la quinta. Por ejemplo: *deus, dei* es de la segunda. Puesto que todas las palabras de la quinta tienen un nominativo que termina en ES, y a ninguna de la segunda le ocurre esto, la identificación de la declinación es fácil. No hay más que comparar TERMINACION con PRIMERA, TERCERA, CUARTA, mediante tests sucesivos y dar el valor apropiado a TERMINACIONES. El inicio de este proceso podría ser:

```
SEA IDENTIFICA
TEST :TERMINACION = :PRIMERA
SIVERDAD [HAZ "TERMINACIONES :TERMI.PRI ALTO]
SIFALSO [TEST :TERMINACION = :TERCERA]
.......................................
```

El TEST para la quinta sería:

```
Y :TERMINACION = :QUINTA PALABRA ULTIMO SINULTIMO :NOM
ULTIMO :NOM = "ES
```

Ten en cuenta, para terminarlo, que tal y como está montado, si la respuesta al test sobre la quinta es falso, seguro que es la segunda. En tal caso tendrás que asignar un nuevo valor a RAIZ. Lo dejamos a tu cargo.

Y ahora, DECLINANDO, que es gerundio. Sabemos la raíz de la palabra que queremos declinar y las terminaciones de la declinación correspondiente que hay que añadirle; conocemos también el nominativo, previsoramente almacenado. ¡Lo sabemos todo!

```
SEA DECLINANDO
(ESCRIBE "NOM :NOMINATIVO)
CONTINUA :CASOS :TERMINACIONES
ESCRIBE []
(TECLEA [¿OTRA? (S/N)] CAR 32)
SI LEECAR="S [DECLINAR]
FIN
```

Por lo que respecta a CONTINUA ten presente que tiene que

procesar las dos listas, CASOS y TERMINACIONES, hasta terminar con ellas. Por supuesto usará RAIZ para escribir :

```
(ESCRIBE PRIMERO :CASOS PALABRA :RAIZ PRIMERO
:TERMINACIONES)
```

Si te parece que no está completo, porque falta el plural, no tienes más que ponerle remedio. Lo único que hay que hacer es...

Conjugar puede parecer más difícil, pero no es imposible. Si te limitas a unos cuantos tiempos simples o, para empezar, a uno solo de los verbos regulares, quizá...

4.8 ¿Necesitas un siquiatra?

¿Puede un programa de ordenador mantener una "conversación" con una persona de modo que ésta no distinga si está comunicándose con un programa o con otra persona? A mediados de los años 60 Weizenbaum mostró, de forma concluyente, que puede hacerse. En su libro *Computer Power and Human Reason* (traducido al castellano con el título *La frontera entre el ordenador y la mente*, Editorial Pirámide, 1978), cuya lectura te recomendamos con entusiasmo, comenta su programa ELIZA. Le dio este nombre porque, como el personaje de Pigmalión, podía aprender a "hablar" cada vez mejor. En el primer experimento con el programa éste podía representar ("parodiar", dice Weizenbaum) el papel de un siquiatra rogeriano celebrando una entrevista inicial con un paciente. Esto era fácil de imitar, ya que gran parte de la técnica del psiquiatra en este caso consiste en obtener información del paciente devolviéndole sus propias frases.

Pero lo que no esperaba su autor fueron las reacciones ante el programa. Una de ellas (te enterarás mejor leyendo el libro) fue que muchos siquiatras en ejercicio consideraron seriamente que el programa podría desarrollarse para utilizarlo en el tratamiento efectivo de pacientes. En opinión de Weizenbaum, esto suponía una absoluta falta de ética profesional, lo que le ocasionó no pocos problemas.

Para que tengas una somera idea de cómo funciona un programa de este tipo vamos a construir el SIQUIATRA, cuya ejecución sería algo parecido a esto:

Cuando el usuario teclea, por ejemplo, el clásico

```
TODO EL MUNDO ME ODIA
```

el programa decide al azar entre dos opciones:

1. Extraer al azar una frase almacenada en una lista que llamaremos FRASES.COMPROMISO, tal como

```
MUCHOS PIENSAN LO MISMO. CONTINUA, POR FAVOR.
```

2. Elegir al azar un principio de frase, de los almacenados en la lista PRINCIPIOS, tal como

```
PARECES CREER QUE
```

al que se añadirá la frase del usuario, con los pronombres y tiempos de los verbos necesarios debidamente cambiados, para producir algo semejante a

```
PARECES CREER QUE TODO EL MUNDO TE ODIA
```

Con estas condiciones podrías escribir:

```
SEA SIQUIATRA
INICIALIZA
ESCRIBE [HOLA, SOY UN SIQUIATRA NO DIRECTIVO]
ESCRIBE [¿CUAL ES TU PROBLEMA?]
RESPUESTAS
FIN

SEA RESPUESTAS
HAZ "FRASE.USUARIO LEELISTA
SI :FRASE.USUARIO = [ADIOS] [ESCRIBE [HASTA OTRA] ALTO]
TEST AZAR 2 = 0
SIVERDAD [ESCRIBE ELIGE :FRASE.COMPROMISO]
SIFALSO [RESPONDE :FRASE.USUARIO]
RESPUESTAS
FIN
```

INICIALIZA debe asignar los valores a FRASE.COMPROMISO, PRINCIPIOS y CAMBIOS. En esta última deberías tener almacenados los cambios de pronombres y verbos que vas a efectuar en la forma en que estaban en el INICIALIZA de TRADUCTOR.

ELIGE lo has manejado ya tantas veces que no nos atrevemos a decir nada.

RESPONDE debe escribir la frase formada por ELIGE :PRINCIPIOS y CAMBIA :FRASE.USUARIO.

CAMBIA debe devolver la frase del usuario con los cambios que se hayan efectuado. Para cada palabra se hace un test para comprobar si está en CAMBIOS.

Si el resultado es VERDAD, se devuelve la frase formada por VALOR :PAL (suponiendo que se haya almacenado en PAL una palabra de la frase), y el resultado de cambiar el resto de la frase del usuario.

Si el resultado es FALSO, se devuelve la frase formada por :PAL y el resultado de cambiar el resto de la frase del usuario.

El programa puede dar bastante juego si el número de frases en FRASE.COMPROMISO es abundante, así como los principios de frase y los cambios.

Si, en lugar del test sobre AZAR 2, escribes

```
HAZ "OPCION AZAR 3
SI :OPCION = 0 [ESCRIBE ELIGE :FRASE.COMPROMISO]
SI :OPCION = 1 [RESPONDE :FRASE.USUARIO]
SI :OPCION = 2 ...
```

puedes añadir otra posibilidad: almacenar de cuando en cuando una de las frases del usuario y "soltarle" posteriormente un

```
ME HAS DICHO ANTES: <frase del usuario>
```

Una forma de utilizar esta última sería

```
SI :OPCION = "2 [RECUERDA :FRASE.REC]
```

donde FRASE.REC puedes inicializarla con [ ] en INICIALIZADA. El procedimiento sería

```
SEA RECUERDA :FRASE.R
TEST VACIAP :FRASE.R
SIVERDAD [HAZ "FRASE.REC :FRASE.USUARIO]
SIFALSO [ESCRIBE FRASE [ME  HAS DICHO ANTES:] :FRASE.REC HAZ
"FRASE.REC :FRASE.USUARIO]
FIN
```

La frecuencia con que el programa dé una u otra no tiene por qué ser siempre la misma.

Puedes poner:

```
HAZ "OPCION AZAR 10
SI MIEMBROP :OPCION [0 1 2 3 4] <opcion 1>
SI MIEMBROP :OPCION [5 6 7] <opcion 2>
SI MIEMBROP :OPCION [8 9] <opcion 3>
```

Y también ampliar las posibilidades de respuesta del programa en otros aspectos. Por ejemplo, puede "reaccionar" ante palabras clave que contenga la frase del usuario; si las palabras "computador", "computadora" o "máquina" aparecen en la frase, podría emitir el mensaje ALGUNAS PERSONAS TEMEN A LAS MAQUINAS. Otro tipo de palabra clave son los "tacos", ante los cuales el mensaje podría ser MODERA TU LENGUAJE, POR FAVOR.

Observa que si las introduces como opciones y resultan seleccionadas, cuando la frase no contenga ninguna palabra clave te verás en un

conflicto. Podrías optar entonces por escribir una frase de compromiso, aunque esto te obligaría a modificar someramente el interior de las opciones.

Piensa con detenimiento los tipos de frases. Prueba el programa con algunos "pacientes" para tener una mejor idea de cuáles son sus frases y después modifícalo teniendo esto en cuenta. Pásale el programa a algún compañero que no esté "metido en el rollo" y escucha su opinión.

4.9 Animal: árboles de información

ANIMAL es un programa muy popular dentro del ámbito de los microordenadores. Se presenta como un juego en el que se pide al usuario que piense en un animal; el programa intentará adivinarlo haciendo preguntas cuya respuesta debe ser *sí* o *no*. Si el programa falla en su intento, pide información al usuario, lo cual produce un aumento de sus "conocimientos". Nos interesa, sobre todo, mostrarte la estructura de los conocimientos del programa: el árbol.

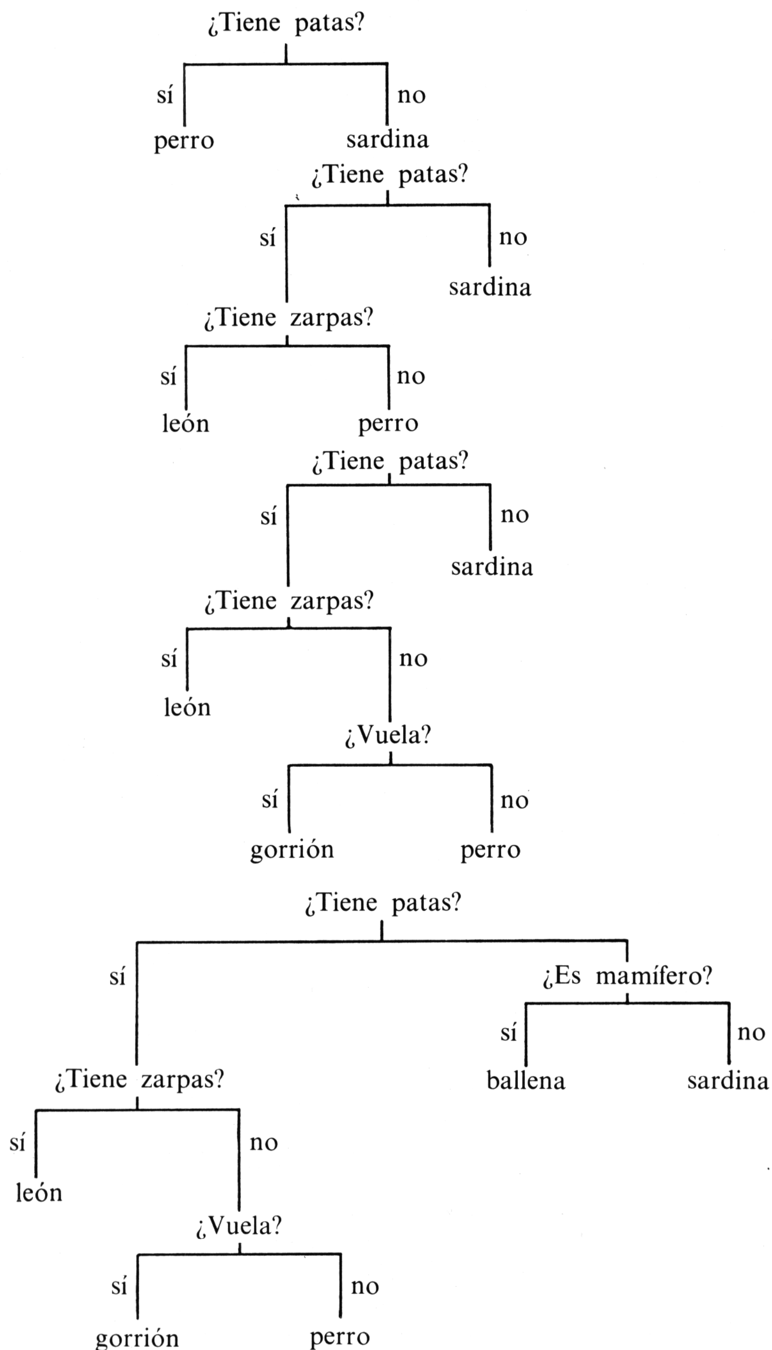
El funcionamiento podría ser algo así:

```
PIENSA EN UN ANIMAL
INTENTARE ADIVINARLO HACIENDOTE PREGUNTAS
¿TIENE PATAS?
SI
TRAS MADURA REFLEXION, EL NOMBRE DEL ANIMAL ES: PERRO
¿HE ACERTADO?
SI
LO SUPONIA. SOY BASTANTE BUENO
¿OTRA VEZ? (S / N)

PIENSA EN UN ANIMAL
INTENTARE ADIVINARLO HACIENDOTE PREGUNTAS
¿TIENE PATAS?
SI
TRAS MADURA REFLEXION, EL NOMBRE DEL ANIMAL ES: PERRO
¿HE ACERTADO?
NO
CUALQUIERA PUEDE EQUIVOCARSE
¿QUE ANIMAL ERA?
LEON
ESCRIBE UNA PREGUNTA CUYA RESPUESTA SEA
SI PARA LEON
NO PARA PERRO
¿TIENE MELENAS?
ENTENDIDO
¿OTRA VEZ? (S / N)
```

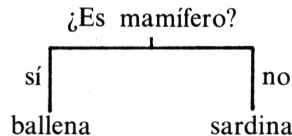
etcétera.

Aquí tienes algunos ejemplos de árboles del conocimiento que podrían ser los de un desarrollo del juego:



Un árbol está formado por nudos y cada uno de ellos tiene dos ramas. A su vez, al final de cada rama puede haber otro nudo con dos ramas, y así sucesivamente. El nudo es la pregunta a responder: si la respuesta es *sí*, se continúa por la rama marcada con el *sí* hasta que se halla un nuevo nudo y se vuelve a preguntar, o bien se encuentra un nombre de animal, en cuyo caso se ha terminado el árbol. Análogamente ocurre con la rama *no*. Observa que todos los terminales son nombres.

Este tipo de organización permite la exploración del conocimiento sin tener que examinarlo todo. Por ejemplo, si en el último árbol la respuesta a "¿tiene patas?" es *no*, sólo hay que preocuparse de explorar la rama *no*, que es



La estructura de árbol no existe como tal en LOGO, de modo que hay que simularla usando listas. Un árbol será una lista con tres elementos: el nudo, la rama *sí* y la rama *no*. El primero de los árboles que has visto sería

```
[[¿TIENE PATAS?] PERRO SARDINA]
```

¿Cuál sería el segundo? Vayamos con cuidado.

El nudo es [¿TIENE PATAS?]

La rama *sí* es a su vez un árbol, de modo que es la lista

[[¿TIENE ZARPAS?] LEON PERRO].

La rama *no* es simplemente SARDINA.

El árbol completo, escrito como lista, es, pues

```
[[¿TIENE PATAS?] [[¿TIENE ZARPAS?] LEON PERRO] SARDINA]
```

Intenta escribir también el tercer árbol como una lista. Si te parece engorroso piensa que luego será el programa quien se encargará de construir los árboles.

Para trabajar con estas estructuras es necesario poder extraer cada uno de los elementos del árbol: nudo, rama *sí* y rama *no*. Si tienes en cuenta que el nudo es siempre el primer elemento de la lista, la rama *sí* el segundo y la rama *no* el tercero, no hay problemas en escribir

procedimientos que devuelvan cada uno de estos elementos en un árbol dado.

```
SEA NUDO :ARBOL
DEVUELVE ITEM 1 :ARBOL
FIN

SEA RAMA.SI :ARBOL
DEVUELVE ITEM 2 :ARBOL
FIN

SEA RAMA.NO :ARBOL
DEVUELVE ITEM 3 :ARBOL
FIN
```

Para comprobarlos escribe:

```
HAZ "CONOCIMIENTOS <alguno de los árboles que tengas>
```

y prueba:

```
ESCRIBE NUDO :CONOCIMIENTOS
ESCRIBE RAMA.SI :CONOCIMIENTOS
ESCRIBE RAMA.NO :CONOCIMIENTOS
```

Si vuelves a leer el ejemplo de funcionamiento del programa observarás que, al principio, conoce el primer árbol que pusimos como ejemplo; por tanto, para empezar a escribirlo podríamos poner:

```
SEA ANIMAL
HAZ "CONOCIMIENTOS [[¿TIENE PATAS?] PERRO SARDINA]
EMPIEZA
FIN

SEA EMPIEZA
(LOCAL "ANIMAL "RESPUESTA)
ESCRIBE [PIENSA EN UN ANIMAL]
ESCRIBE [INTENTARE ADIVINARLO HACIENDOTE PREGUNTAS]
ELIGE.RAMA  :CONOCIMIENTOS  ;[SE ENCARGA DE IR EXPLORANDO EL
ARBOL HASTA DAR UN VALOR A ANIMAL]
ESCRIBE [TRAS MADURA REFLEXION, EL NOMBRE]
(ESCRIBE [DEL ANIMAL ES:] :ANIMAL)
HAZ "RESPUESTA RESPONDE [¿HE ACERTADO?]
TEST :RESPUESTA = "SI
SIVERDAD [ESCRIBE [LO SUPONIA. SOY BASTANTE BUENO]
SIFALSO [APRENDERE]
HAZ "RESPUESTA RESPONDE [¿OTRA VEZ?]
SI :RESPUESTA = "SI [EMPIEZA] [ALTO]
FIN
```

Suponemos que no sobrarán algunos comentarios. El fragmento ;[SE ENCARGA DE IR EXPLORANDO EL ARBOL HASTA DAR UN VALOR A ANIMAL] es un comentario para aclarar al autor del programa, o a quien lo lea, lo que hace ELIGE.RAMA. Puesto que

LOGO no dispone de ninguna primitiva que permita incluir comentarios, hay que fabricarla. Por ejemplo:

```
SEA ; :COMENTARIO
FIN
```

Como ves, el nombre del procedimiento es ";" y su entrada, una lista cualquiera. Ingenioso, ¿no?

El procedimiento RESPONDE lleva incorporado como entrada la pregunta a la que hay que responder, y debe filtrar la respuesta devolviendo solamente SI o NO. Esto quiere decir que si el usuario contesta DESDE LUEGO, el procedimiento debe pedirle amablemente que se limite a escribir SI o NO. Lo dejamos a tu iniciativa.

ELIGE.RAMA es relativamente fácil de hacer si recuerdas cómo se obtienen el nudo (la pregunta) y las ramas *sí* y *no* del árbol. Tras cada respuesta, el procedimiento debe llamarse a sí mismo eligiendo la rama correspondiente. Al llegar a un nombre de animal, que es una palabra y no una lista, la búsqueda ha terminado. Qué te parece esto:

```
SEA ELIGE.RAMA :ARBOL
SI PALABRAP :ARBOL [HAZ "ANIMAL :ARBOL ALTO]
HAZ "RESPUESTA RESPONDE NUDO :ARBOL
TEST :RESPUESTA = "SI
SIVERDAD [ELIGE.RAMA RAMA.SI :ARBOL]
SIFALSO [ELIGE.RAMA RAMA.NO :ARBOL]
FIN
```

Por lo que respecta a APRENDERE, teniendo en cuenta el comportamiento del programa, debería ser algo así:

```
SEA APRENDERE
(LOCAL "ANIM.CORRECTO "PREGUNTA)
ESCRIBE [CUALQUIERA PUEDE EQUIVOCARSE]
ESCRIBE [¿QUE ANIMAL ERA?]
HAZ "ANIM.CORRECTO PRIMERO LEELISTA
ESCRIBE [ESCRIBE UNA PREGUNTA CUYA RESPUESTA SEA]
(ESCRIBE [SI PARA] :ANIM.CORRECTO)
(ESCRIBE [NO PARA] :ANIMAL)
HAZ "PREGUNTA LEELISTA
 <darle un nuevo valor a CONOCIMIENTOS reemplazando en él
:ANIMAL por la lista formada por :PREGUNTA, :ANIM.CORRECTO y
:ANIMAL>
FIN
```

La última línea puedes escribirla de esta manera:

```
HAZ "CONOCIMIENTOS REEMPLAZA :CONOCIMIENTOS :ANIMAL (LISTA
:PREGUNTA :ANIM.CORRECTO :ANIMAL)
```

Lo que queda por hacer es escribir un procedimiento que reemplace en un árbol una rama por otra.

```
SEA REEMPLAZA :ARBOL :RAMA :NUEVA.RAMA
```

El procedimiento deberá devolver el árbol nuevo, es decir, una lista formada por la pregunta inicial y el resultado de reemplazar recursivamente en cada una de las ramas del árbol. Precisemos: la lista formada por los tres elementos siguientes:

```
NUDO :ARBOL
REEMPLAZA RAMA.SI :ARBOL :RAMA :NUEVA.RAMA
REEMPLAZA RAMA.NO :ARBOL :RAMA :NUEVA.RAMA
```

Si consideras el paso del primer árbol al segundo, la ejecución del procedimiento se va a encontrar con

> REEMPLAZA en el árbol PERRO la rama PERRO por [[¿TIENE ZARPAS] LEON PERRO]

en cuyo caso debe devolver la nueva rama,

> [[¿TIENE ZARPAS?] LEON PERRO]

y también con

> REEMPLAZA en el árbol SARDINA la rama PERRO por [[¿TIENE ZARPAS?] LEON PERRO]

lo que debería devolver el árbol

> SARDINA.

Así pues, las dos primeras instrucciones de REEMPLAZA serían

```
SI :ARBOL = :RAMA [DEVUELVE :NUEVA.RAMA]
SI PALABRAP :ARBOL [DEVUELVE :ARBOL]
```

precisamente en este orden (¿por qué no al reves?), seguidas de

```
DEVUELVE (LISTA <los tres elementos indicados anteriormente>)
```

Escribe el programa completo y pruébalo. Comprobarás que acorrala rápidamente al jugador. Al sentirse acorralado, y a veces antes, los hay que repiten un nombre de animal que ya han dado. Prueba y verás cómo APRENDERE no aprende tanto. Cuando ocurre algo así, el que ha escrito el programa suele sentirse un poco frustrado. ¿Podrías corregirlo para que no admitiera nombres duplicados?

A veces el jugador empieza a dar nombres de animales imaginarios y, para distinguirlos, ofrece preguntas más bien fantásticas. Ello no le causa pavor a nuestro programa, y el juego suele resultar divertido.

Con unas cuantas reformas en ANIMAL puedes conseguir que empiece con los conocimientos que adquirió la última vez, así como que almacene en discos los conocimientos adquiridos, o que no lo haga, como desees. Recuerda para ello lo que has hecho en otros programas que aprenden.

Ten en cuenta que ANIMAL es, esencialmente, un almacén de información en forma de árbol binario y que su estructura puede aplicarse en otros muchos casos. Piensa en alguno y organiza el juego correspondiente.

Mostrando la información

Podrías pensar en algún procedimiento que te permita escribir toda la información que hay almacenada, en la forma siguiente (para el último de los árboles del ejemplo):

LEON [TIENE ZARPAS] [TIENE PATAS]
GORRION [VUELA] [NO TIENE ZARPAS] [TIENE PATAS]
PERRO [NO VUELA] [NO TIENE ZARPAS] [TIENE PATAS]
BALLENA [ES MAMIFERO] [NO TIENE PATAS]
SARDINA [NO ES MAMIFERO] [NO TIENE PATAS]

La cabecera de este procedimiento podría ser

```
SEA EXTRAE :ARBOL :LISTA.PROP
```

Lo que tendría que hacerse es:

Poner el nudo del árbol el último de LISTA.PROP.
Extraer la información de la rama *sí* del árbol con el valor actual de LISTA.PROP.

(Mientras estas llamadas recursivas se vayan produciendo se irá acumulando en LISTA.PROP la lista de propiedades. Cuando el árbol quede reducido a una palabra es que se ha terminado de inspeccionar la rama y entonces hay que escribir el árbol —que será el nombre del animal— y la lista de propiedades, deteniéndose el procedimiento.)

Extraer la información de la rama *no* del árbol.

Para ello ten presente que al terminar la rama *sí* te encontrarías, por ejemplo, con un valor de LISTA.PROP tal como [TIENE PATAS] [TIENE ZARPAS], que son las propiedades de LEON. Habría

que extraer ahora de la rama *no,* pero no con este valor para LISTA .PROP, sino poniendo

```
HAZ  "LISTA.PROP PONULTIMO (FRASE "NO NUDO :ARBOL) SINULTIMO
:LISTA.PROP
```

ya que de la lista hay que quitar [TIENE ZARPAS] para poner [NO TIENE ZARPAS]

Estarás pensando que si NUDO es el procedimiento que conoces, el resultado de NUDO :ARBOL va a tener los signos de interrogación. Tienes toda la razón: fabrícate un NUDO1, para usar en EXTRAE, que suprima estos signos.

No olvides escribir EXTRAE y déjalo junto con ANIMAL. Cuando hayas jugado con él y lo tengas atiborrado de información, efectúa una llamada escribiendo:

```
EXTRAE :CONOCIMIENTOS []
```

para comprobar la información almacenada.

4.10 El NIM

Existen varias versiones de este juego chino. Una de las más sencillas es la siguiente:

De un montón de 21 objetos dos jugadores retiran, por turno, un mínimo de uno y un máximo de tres; el que se vea obligado a retirar el último, pierde.

La idea (debida a Papert y Solomon) es desarrollar un programa imbatible procediendo por etapas, esto es, construyendo programas cada vez "más listos".

El primer programa es simplemente "un marcador" que se limita a llevar la cuenta de las que quita cada jugador y de las que quedan. Una partida podría tener este aspecto:

```
EL NUMERO DE OBJETOS ES 21
JUEGA PEDRO ¿CUANTAS QUITAS?
3
EL NUMERO DE OBJETOS ES 18
JUEGA PABLO ¿CUANTAS QUITAS?
3
```

y así sucesivamente.

Es fácil escribir un procedimiento recursivo que tenga como entradas el número de objetos y los nombres de los jugadores. Aquí está:

```
SEA NIM.MARCADOR :OBJETOS :JUGADOR1 :JUGADOR2
ESCRIBE FRASE [EL NUMERO DE OBJETOS ES] :OBJETOS
ESCRIBE (FRASE "JUEGA :JUGADOR1 [¿CUANTAS QUITAS?])
HAZ "QUEDAN :OBJETOS - QUITAS
NIM.MARCADOR :QUEDAN :JUGADOR2 :JUGADOR1
FIN
```

QUITAS es un un procedimiento que devuelve el número tecleado por el jugador; acuérdate de LEENUMERO.

Compara la llamada recursiva con la cabecera. ¿Verdad que es astuto?

Si ensayas un poco el procedimiento observarás que:

- Cada jugador puede quitar las que quiera, no de 1 a 3.
- Cuando se ha quitado la última el programa sigue como si el juego no hubiese terminado.

El NIM.ARBITRO se va a encargar de resolver esos problemas. El primero se consigue modificando QUITAS de manera que sólo admita números entre 1 y 3, cosa fácil usando MIEMBROP. ¿Estás seguro de que esto lo resuelve todo?

Detener el juego tampoco presenta problemas. Si QUEDAN es 0, el juego ha terminado, siendo el ganador el nombre almacenado en ese momento en JUGADOR2, ya que el otro ha retirado la última. Debe emitirse el mensaje oportuno sobre el ganador.

Escribe en detalle NIM.ARBITRO. Si lo deseas puedes empezar con una introducción que dé las instrucciones del juego y pregunte los nombres de los jugadores, antes de lanzar el procedimiento recursivo. Compruébalo.

La etapa siguiente ya te la estás imaginando. Uno de los jugadores puede ser el propio programa. Las modificaciones importantes estarían en el procedimiento QUITAS, que ahora debe saber a quién le toca jugar, y el número de objetos que hay, si el programa quiere utilizar una buena estrategia. Es decir, la cabecera debe ser

```
QUITAS :JUGADOR :OBJETOS
```

Si el jugador no es el programa, todo se desarrollará como antes, y si es el programa..., bueno, siempre queda una solución: elegir un número al azar y esperar que la suerte sea propicia. Posiblemente te interese escribir procedimientos para las acciones a realizar en QUITAS, según le toque al programa o al jugador. Si no se te ha ocurrido nada mejor que el azar, escribe en detalle NIM.PROGRAMA.B y pruébalo. Entonces te percatarás de que B podría ser la inicial de bobo; en efecto, puede ocurrir que queden 2 ó 3 y el programa las retire todas, en lugar de retirar una o dos.

El paso a NIM.PROGRAMA requiere simplemente que sustituyas la elección al azar por una estrategia. Hay dos formas, que no se excluyen mutuamente, de determinarla. Una es jugar con la versión B,

anotando los resultados de las partidas y estudiándolos. La otra consiste en empezar el estudio, lápiz en mano, por el final de la partida.

Si quedan 2, 3 ó 4 se gana seguro quitando 1, 2 ó 3, respectivamente.

Si quedan 6, 7 u 8 se gana seguro quitando 1, 2 ó 3, respectivamente, ya que dejándole 5 al contrario, haga lo que haga, dejará 2, 3 ó 4.

Si quedan 10, 11 ó 12 se gana seguro quitando 1, 2 ó 3, respectivamente, ya que dejándole 9 al contrario, haga lo que haga, dejará 6, 7 u 8.

Así pues, si quedan N objetos, para ganar hay que quitar...

La cosa no estaría completa sin unas instrucciones para los números perdedores. Si quedan 5, 9, etc., que son números perdedores, quizá lo más prudente es quitar 1. Si el contrario juega con estrategia no va a servir de nada, pero si el contrario es humano, ya se sabe que *errare humanum est*, ¡si se equivoca en sus cálculos está perdido!

Observa que si el programa juega con la estrategia óptima, sabiendo cuáles son los números perdedores, puede avisar al jugador, antes de terminar el juego, que ya está perdido. El personal suele quedar anonadado en casos así. Incluye este refinamiento si tienes la estrategia.

Los números ganadores y perdedores a que nos hemos referido antes valen en el caso de que el máximo a quitar sea 3. ¿Y si se puede quitar otro número?

Si has escrito el programa "inteligente" e invitas a otros a jugar con él, en la introducción podrías preguntar el nombre del jugador y hacer que el programa eligiera el número de objetos con los que se va a jugar, así como el número máximo de los que se pueden quitar.

Seguro que se te ocurren otros perfeccionamientos. ¿Dibujar los objetos? Ensáyalos.

4.11 Instrucciones de programación estructurada

Si has programado en PASCAL o en otro lenguaje estructurado, la recursión no te resultará extraña, pero echarás de menos instrucciones tales como MIENTRAS-HAZ, REPITE-HASTA, DEL-AL-HAZ. Estas no existen en LOGO como primitivas, pero pueden fabricarse procedimientos que realicen su trabajo.

Supón que necesitas una instrucción que escriba un número y lo incremente mientras sea menor que 10, que en LOGO se escribiría:

```
MIENTRAS [:X < 10] [ESCRIBE :X HAZ "X :X+1]
```

donde la primera lista contiene la condición que debe cumplirse y la segunda las instrucciones a ejecutar mientras la condición se cumpla. El procedimiento MIENTRAS sería, según esto,

```
SEA MIENTRAS :CONDICION :INSTRUCCION
SI NO (EJECUTA :CONDICION) [ALTO]
EJECUTA :INSTRUCCION
MIENTRAS :CONDICION :INSTRUCCION
FIN
```

Para una instrucción REPITE-HASTA podrías poner

```
REPITE [ESCRIBE :X HAZ "X :X+1] [:X > 9]
```

siendo el procedimiento

```
SEA REPITE :INSTRUCCION :CONDICION
EJECUTA :INSTRUCCION
SI EJECUTA :CONDICION [ALTO]
REPITE :INSTRUCCION :CONDICION
FIN
```

Una instrucción DEL-AL-HAZ que escriba números del 1 al 9 podría ser:

```
DEL.AL "X 1 9 [ESCRIBE :X]
```

siendo el procedimiento

```
DEL.AL :VARIABLE :INICIAL :FINAL :INSTRUCCION
SI :INICIAL > :FINAL [ALTO]
HAZ :VARIABLE :INICIAL
EJECUTA :INSTRUCCION
DEL.AL :VARIABLE :INICIAL+1 :FINAL :INSTRUCCION
FIN
```

A lo mejor te parece que puede haber problemas si se trata de agrupar instrucciones de este tipo. Por ejemplo, para escribir todos los números de tres cifras, distintas o no, que pueden formarse con los dígitos 1 al 5, podrías escribir

```
DEL.AL "X 1 5 [DEL.AL "Y 1 5 [DEL.AL "Z 1 5 [(TECLEA  :X  :Y
:Z) ESCRIBE "]]]
```

No es tan difícil.

4.12

El juego de la vida

¿Cómo evoluciona una población a lo largo del tiempo, dadas unas condiciones sobre nacimientos y muertes? De entre las múltiples posibilidades de reglas para regular estos aspectos vamos a comentar una de las primeras en hacerse realmente populares: la propuesta por el biólogo Horton Conway en 1970 con el nombre de "juego de la vida".

Comencemos con un "universo" formado por un tablero de 9 por 9 casillas, en cada una de las cuales puede vivir una célula. Dada una determinada distribución de las células sobre el tablero, la "generación siguiente" se forma de acuerdo con estas reglas:

1. Son vecinas de una célula todas las que están en los ocho cuadrados, como máximo, que tienen un lado o un vértice común con aquél donde reside la célula.
2. Si un cuadrado contiene una célula viva, ésta continuará viva en la siguiente generación, si tiene dos o tres vecinas. En otro caso muere, de soledad o superpoblación.
3. Si un cuadrado está vacío, en él nacerá una célula en la siguiente generación, si tiene exactamente tres vecinas; en otro caso permanecerá vacío.

Comprueba que con estas reglas se tiene la siguiente sucesión de generaciones:

```
                    *                           *
    ***             *            ***            *
                    *                           *
```

generación 1   *generación 2*   *generación 3*   *generación 4*

Como ves, esta configuración es estable. Pero ¿qué ocurre con otras? ¿Se estabilizan, crecen, desaparecen? Vamos a construir un programa que nos permita el estudio de la evolución de estas poblaciones.

Lo primero es elegir una representación para el "universo". Cada cuadrado del universo podría representarse por un par de números, fila y columna, ambos variando de 1 a 9, y, puesto que hay que manejar dos generaciones consecutivas, actual y siguiente, podríamos utilizar una letra para distinguirlas, de modo que A21 significaría: segunda fila, primera columna, de la generación actual; S35 significaría: tercera fila, quinta columna, de la generación siguiente.

Las variables de este tipo, con índices, no existen como tales en LOGO, pero pueden construirse. A21 puede ser una palabra en LOGO; por tanto,

```
(PALABRA "A "2 "1)
```

resuelve el problema. Si suponemos que la letra está almacenada en la variable ACT, la fila y la columna en las variables FIL y COL, haciendo variar éstas, la instrucción:

```
(PALABRA :ACT :FIL :COL)
```

nos permitirá representar todas las posiciones del tablero para la generación actual. ¿Cómo indicamos que una posición contiene una célula viva? Podemos asignar el valor VERDAD a la posición correspondiente, escribiendo

```
HAZ (PALABRA :ACT :FIL :COL) "VERDAD
```

¿Cuáles son las vecinas de la que reside en la fila F, columna C? Las que residen en las filas comprendidas entre :F – 1 y :F + 1 y en las columnas comprendidas entre :C – 1 y :C + 1, excluyendo por supuesto a la propia célula. ¿Seguro?

En el caso de las filas y columnas primera y última, el criterio anterior dará vecinos "fuera del universo". No es un grave problema, se filtra o se crean "falsos" vecinos. Volveremos sobre el asunto.

Para pasar de una generación a la siguiente basta aplicar las reglas a todo el tablero. Suponiendo que has inicializado FILFINAL y COLFINAL a 9 y FYCINI (fila y columna inicial) a 1, para cada posición de la generación actual hay que ver si la célula está viva, haciendo un test sobre el valor, VERDAD o FALSO, de la posición. Según sea el caso hay que contar el número de vecinos asignando el valor, de acuerdo con las reglas, a la generación siguiente. Aquí tienes una posibilidad:

```
SEA CAMBIA :ACT :SIG :FIL :COL
(LOCAL "ACTUAL "SIGUIENTE)
SI :COL > :COLFINAL [HAZ "COL :FYCINI HAZ "FIL :FIL + 1]
SI :FIL > :FILFINAL [ALTO]
HAZ "ACTUAL (PALABRA :ACT :FIL :COL)
HAZ "SIGUIENTE (PALABRA :SIG :FIL :COL)
TEST VALOR :ACTUAL
SIVERDAD [SI MIEMBROP VECINOS :ACT :FIL :COL [3 4] [HAZ
"SIGUIENTE "VERDAD] [HAZ "SIGUIENTE "FALSO]]
SIFALSO [SI VECINOS :ACT :FIL :COL = "3 [HAZ "SIGUIENTE
"VERDAD] [HAZ "SIGUIENTE "FALSO]]
CAMBIA :ACT :SIG :FIL :COL + 1
FIN
```

Obviamente VECINOS debe devolver el número de vecinas vivas de cada posición. Observarás que hay una unidad más de las indicadas por las reglas: es para no molestarse en excluir de la cuenta a la propia célula. Utilizando las variables locales NUMERO, FI (fila inicial), FF (fila final), CI y CF podrías escribir

```
SEA VECINOS :GEN :F :C
(LOCAL "NUMERO "FI "FF "CI "CF)
```

```
HAZ "NUMERO 0
HAZ "FI :F - 1
HAZ "FF :F + 1
HAZ "CI :C - 1
HAZ "CF :C + 1
CONTAR :GEN :FI :CI
DEVUELVE :NUMERO
FIN
```

CONTAR se limita a examinar las posiciones, empezando con FI, CI y terminando con FF, CF, aumentando el valor de NUMERO en una unidad si la célula que ocupa esa posición está viva. Y aquí pueden surgir los problemas que antes señalábamos. Si estás investigando los vecinos de A15, por ejemplo, tres de ellos son A04, A05 y A06, que están fuera del universo y, por tanto, no han recibido ningún valor. Una instrucción del tipo

```
SI VALOR (PALABRA :G :F :C) [HAZ "NUMERO :NUMERO + 1]
```

que podría formar parte del procedimiento CONTAR :G :F :C, produciría la ruptura del programa con la emisión de un mensaje indicando que la variable no tiene valor.

Tienes dos opciones para resolver el problema. Una es impedir que ocurra, escribiendo:

```
HAZ "FI :F - 1
SI :FI < FYCINI [HAZ "FI :FYCINI]
HAZ "FF :F + 1
SI :FF >FILFINAL [HAZ "FF :FILFINAL
```

y análogamente para las columnas.

Otra es crear "falsos vecinos". Al inicializar las variables A y S, en lugar de hacerlo desde 1 a 9 se hace desde 0 a 10, es decir, de :FYCINI – 1 a :FILFINAL + 1. y :COLFINAL + 1.

¿Cuál te parece la mejor solución? No contestes a la ligera. La mejor solución será la que dé una mayor rapidez al proceso de cambio, lo que permitirá observar más generaciones por unidad de tiempo.

Una vez que tienes la siguiente generación, es cuestión de representarla en pantalla y volver a repetir, pero ahora SIG representa a la generación actual y habrá que "limpiar" ACT, inicializándola de nuevo, para que tome el papel de siguiente. Utilizando la variable NUM para llevar la cuenta del número de generación, podría escribirse:

```
SEA VIDA :ACT :SIG :NUM
CAMBIA :ACT :SIG :FYCINI :FYCINI
PANTALLA :SIG :FYCINI :FYCINI
ESCRIBE FRASE [GENERACION] :NUM
INI :ACT :FYCINI :FYCINI
VIDA :SIG :ACT :NUM + 1
FIN
```

Puesto que es necesario inicializar las variables, así como introducir una población inicial, se podría pensar en

```
SEA VIDAS
LT INICIALIZA
POBLAINI
LT PANTALLA "A :FYCINI :FYCINI
ESCRIBE [GENERACION 1]
VIDA "A "S 1
FIN
```

Es el momento de hacer otra observación, no la última, sobre la rapidez. El procedimiento PANTALLA va a limitarse a escribir un "*" en una determinada posición, si la célula está viva, y un espacio en blanco, CAR 32, si no lo está. Si esto se hace después de CAMBIA, el tiempo de espera va a parecer muy grande. Psicológicamente sería mejor ver signos de actividad según se vaya produciendo el cambio, cosa que puede conseguirse sólo con escribir

```
CURSOR LISTA :FIL :COL
SI VALOR :SIGUIENTE [TECLEA "*] [TECLEA CAR 32]
```

antes de la llamada recursiva de CAMBIA. Esto haría innecesario el procedimiento PANTALLA, que puedes borrar de VIDA ahora mismo; así como de VIDAS. POBLAINI, que recoge la población inicial, se encargaría de poner las células de la primera generación en el tablero. Si utilizas la parte superior de la pantalla para representar el tablero, convendría pedir las coordenadas y escribir la generación en una fila "alta", la 23 por ejemplo, de modo que en VIDA, antes de escribir el número de generación, debería incluir un

```
CURSOR [23 0]
```

Por lo que respecta a VIDAS, podría quedar así:

```
SEA VIDAS
LT INICIALIZA
POBLAINI
CURSOR [23 0] ESCRIBE [GENERACION 1]
VIDA "A "S 1
FIN
```

Vamos a proporcionar algunas indicaciones sobre el par de procedimientos que restan. INICIALIZA debe dar valores a FYCINI, FILFINAL y COLFINAL, así como llamar a INI "A :FYCINI – 1 :FYCINI – 1 e INI "S :FYCINI – 1 :FYCINI – 1, para que asignen valores a todas las posiciones de las generaciones; también debería llamar a BORDES, un procedimiento que escribiese algo semejante a

```
 123456789
1
2
3
4
5
6
7
8
9
```

(en la pantalla de LOGO te quedará más cuadrado), para que puedas observar cuáles son las posiciones de las células y los límites del "universo". Ya puedes empezar a trabajar sobre lo que queda a tu cargo.

POBLAINI debe pedir las coordenadas de cada célula y representarla en la pantalla, en el cuadrado que BORDES ha dejado preparado, deteniéndose cuando se pulse retorno sin escribir nada.

```
SEA POBLAINI
(LOCAL "COOR "COORF "COORC)
CURSOR [23 0]
(TECLEA [¿COORDENADAS?] CAR 32)
HAZ "COOR LEELISTA
TEST COOR = []
SIVERDAD [ALTO]
SIFALSO [<asignar VERDAD a la posición correspondiente de la
generación A y escribirla en la pantalla>]
POBLAINI
FIN
```

La sorpresa que puedes llevarte consiste en que, cuando vayas a escribir las segundas coordenadas, las primeras están aún escritas (después se pondría en ese lugar GENERACION 1, etc.), de modo que es conveniente que limpies la línea antes de volver a escribir. Limpiar la línea significa escribir blancos en las posiciones ocupadas, no más de 20, para lo cual podrías usar el procedimiento

```
SEA BORRALINEA
SI :C > 20 [ALTO]
CURSOR LISTA :F :C
TECLEA CAR 32
BORRALINEA :F :C+1
FIN
```

Estudia bien el lugar donde piensas insertarlo. Si lo emplazas adecuadamente sólo necesitarás hacerlo una vez.

Pon en marcha el programa con una población de células cuyo comportamiento conozcas (utiliza la del ejemplo del principio). Observarás que el cambio de generación es lento, debido a que el cursor se pasea por zonas de la pantalla en las que ni hay ni habrá ninguna célula. Convendría que sólo examinase las posiciones necesarias, sobre todo si en lugar de 9*9 el tablero es de 19*19.

Antes de continuar, un problema que plantea el uso de dos dígitos: ¿qué significa A114? Está claro: la fila 1, columna 14, ¿o tal vez la fila 11, columna 4? Con A11_4 y A1_14 no existe esa pega, de modo que revisa ahora mismo todas las veces que has usado PALABRA para construir variables con índices.

Continuemos. Si has utilizado como población inicial las células [3 4] [3 5] [4 4] [4 5] [4 6] convendría que el programa determinase automáticamente que las células están en el rectángulo que va de la fila 3 a la 4 y de la columna 4 a la 6. Así, CAMBIO no examinaría todo el tablero, sino el rectángulo que va desde las filas 2 a 5 y las columnas 3 a 7 (¿por qué hemos ampliado en una unidad los lados del rectángulo anterior?). Siempre que se produzca un cambio de generación hay que fijar nuevos límites.

Una vez recibida la primera célula, el procedimiento PONLIMITES debería dar a las variables FIL.INF, FIL.SUP, COL.INF, COL.SUP los valores 3, 3, 4, 4, respectivamente. Al recibir la segunda deberían ser 3, 3, 4, 5, respectivamente; por tanto, el nuevo valor de COL.INF es el mínimo de su anterior valor y el de la nueva coordenada, mientras que el de COL.SUP es el máximo de su anterior valor y de la nueva coordenada. Al entrar [4 4] deberían ser 3, 4, 4, 5, con lo cual, para las filas mínima y máxima, servirán las mismas relaciones. Compruébalo con las dos que restan.

Evidentemente habría que dar unos valores iniciales a las variables, cosa de la que se encargaría INILIMITES. Si asignas a las inferiores el de FYCINI y a las superiores los de FILFINAL y COLFINAL, entonces no valen las relaciones, ya que al entrar [3 4] se obtendría 1, 1, 9, 9, que no son los valores deseados. El problema reside en los valores iniciales, y la solución es... ¡cambiarlos! Comprueba que si pones

```
SEA INILIMITES
HAZ "FIL.INF :FILFINAL
HAZ "FIL.SUP :FYCINI
HAZ "COL.INF :COLFINAL
HAZ "COL.SUP :FYCINI
FIN
```

y aplicas las relaciones que hemos visto antes, es decir,

```
SEA PONLIMITES :F :C
HAZ "FIL.INF MIN :FIL.INF :F
HAZ "FIL.SUP MAX :FIL.SUP :F
HAZ "COL.INF MIN :COL.INF :C
HAZ "COL.SUP MAX :COL.SUP :C
FIN
```

todo marcha sobre ruedas. MAX es un procedimiento que devuelve la mayor de sus dos entradas. MIN te lo puedes imaginar.

Vamos a ir revisando los procedimientos para advertir cómo hay que modificarlos para construir VIDAS1.

En INICIALIZA deberías incluir un INILIMITES antes de poner en marcha los INI. Dentro de este procedimiento los INI van a dar valores a A y S utilizando fila y columna inicial, fila final y columna final, pero posteriormente sólo modificarán los valores marcados por los límites, de modo que deberías incluir en ellos más parámetros; algo parecido a

```
SEA INI :GEN :F :C :FI :FF :CI :CF
```

y la llamada para A dentro de INICIALIZA sería

```
INI  "A  :FYCINI-1  :FYCINI-1  :FYCINI-1 :FYCINI-1 :FILFINAL
:COLFINAL
```

En POBLAINI, tras la recepción de cada célula, tendrás que usar un PONLIMITES. ¿Te parece bien al final de la opción SIFALSO del test?

En VIDA, que lanza a CAMBIA, parece que deberías llamar a éste escribiendo

```
CAMBIA :ACT :SIG :FIL.INF :COL.INF
```

En CAMBIA, teniendo en cuenta los nuevos límites, tendrías que cambiar FYCINI, FILFINAL y COLFINAL por FIL.INF, COL .INF, FIL.SUP, COL.SUP, e incluir PONLIMITES cada vez que nace una célula, esto es, siempre que tengas un HAZ "SIGUIENTE "VERDAD.

Si lo piensas un poco deducirás que esto no funciona, ya que si cambias los límites en cada nacimiento pierdes los de la generación que estás estudiando. La solución está en utilizar otras variables para guardar los límites de la generación que estudias y calcularlos para la nueva; de manera que VIDA sería:

```
SEA VIDA :ACT :SIG :NUM
HAZ "F.INF :FIL.INF - 1
HAZ "F.SUP :FIL.SUP + 1
HAZ "C.INF :COL.INF - 1
HAZ "C.SUP :COL.SUP + 1
SI :F.INF < :FYCINI [HAZ "F.INF :FYCINI]
SI :F.SUP > :FILFINAL [HAZ "F.SUP :FILFINAL]
<lo mismo para las columnas>
INILIMITES
CAMBIA :ACT :SIG :F.INF :C.INF
<lo que sigue es cosa tuya>
```

y en CAMBIA tendrás que sustituir FYCINI, FILFINAL y COLFINAL por F.INF, C.INF, F.SUP, C.SUP, incluyendo, por supuesto, PONLIMITES donde dijimos anteriormente.

Escribe, pues, cuidadosamente VIDAS1 y compáralo con VIDAS. Aumenta el tablero a 15, por ejemplo. Para ello sólo tienes que

modificar los valores de FILFINAL y COLFINAL en INICIALIZA.

Desde luego esto no termina aquí; las posibilidades de variaciones son inmensas:

¿Qué sucede si las células no mueren de soledad?

¿Y si sólo se producen nacimientos cuando haya dos vecinas?

¿Qué pasa si los nacimientos y las muertes no ocurren "siempre", sino con unas ciertas probabilidades?

¿Cómo se comportarán las poblaciones si sólo se considera vecinas a las que están en los cuatro cuadrados que tienen un lado común con aquél donde reside la célula?

Con estas interrogantes ya tienes para entretenerte un buen rato, pero si quieres más, ahí va otra idea:

Imagina en el tablero dos especies en competencia. Proporciona las condiciones para nacimientos y muertes de una y otra, coloca algunos individuos sobre el tablero y... a ver qué pasa.

No seguimos. Quizá en Ciencias Naturales encuentres ideas sobre modelos de este tipo. La Ecología está llena de ellos.

4.13 Cuerpos celestes

Al hablar de la tortuga dinámica vimos cómo cambiar su posición conociendo su vector velocidad, y cómo cambiar éste conociendo su vector aceleración. Sabiendo esto es fácil representar en pantalla los movimientos de dos cuerpos celestes. Los vectores aceleración pueden calcularse de manera que, obtenido uno de ellos, el otro sea el opuesto multiplicado por una constante. Tu profesor de Física estará encantado de proporcionarte los detalles.

En última instancia puedes utilizar el siguiente procedimiento como una "caja negra". Necesita como entradas POS1 y POS2, posiciones de los cuerpos en forma de lista, y la constante ATRACCION, que es un número; y devuelve el vector aceleración como una lista.

```
SEA ACEL.MOV
(LOCAL "D "V)
HAZ "D DISTANCIA :POS2 :POS1
HAZ "V DIFE.LISTAS :POS2 :POS1
DEVUELVE MULTIPLICA (:ATRACCION/(:D*:D*:D)) :V
FIN
```

DISTANCIA y DIFE.LISTAS ya los conoces. El último devuelve la lista diferencia de las dos listas de entrada.

MULTIPLICA :NUMERO :VECTOR, en el que VECTOR es una lista, debe devolver una relación cuyos elementos son los de la lista de entrada multiplicados por el valor de NUMERO.

Ya podemos empezar:

```
SEA CUERPOS.CELESTES
INICIALIZA
LG ET VENTANA
MUEVELOS
FIN
```







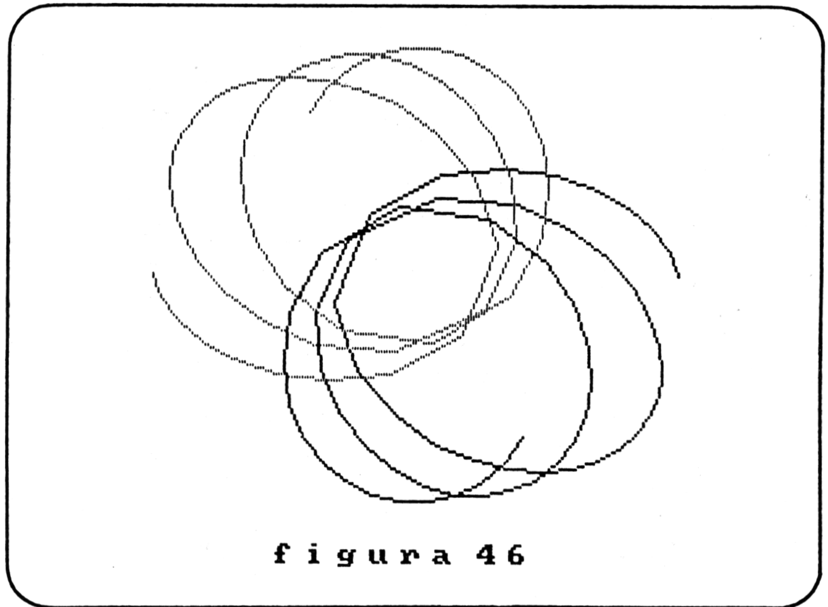


figura 46

Te sugerimos este INICIALIZA:

```
SEA INICIALIZA
HAZ "POS1 [-100 0]
HAZ "VEL1 [1 -10]
HAZ "COLOR1 1
HAZ "POS2 [100 0]
HAZ "VEL2 OPUESTO :VEL1
HAZ "COLOR2 2
HAZ "ATRACCION 100000
FIN
```

Se supone que no tendrás ningún inconveniente para escribir el procedimiento OPUESTO, cuya entrada es una lista y cuya salida es la correspondiente al vector opuesto al de la lista de entrada. Los colores corresponden a IBM (en APPLE usa 1 y 5).

Si no te gusta cómo trabaja, introduce los cambios que te parezcan.

MUEVELOS debe dejar la traza de dos "cuerpos" con una sola tortuga. Esto quiere decir que tendrá que dibujar un trocito de la trayectoria del cuerpo 1, luego otro de la del cuerpo 2, y así sucesivamente. Para cada "cuerpo" habrá que actuar del modo siguiente:

Levantar el lápiz para que el paso de uno a otro no deje traza.
Asignar el color con el que se va a dibujar.
Colocar el cuerpo (la tortuga) en la posición donde se quedó la última vez que se movió.
Desplazarle teniendo en cuenta que, al depender la acelaración de las posiciones de los dos cuerpos, hay que calcularla cada vez.
Guardar la posición y la velocidad actuales para la próxima ocasión.

¿Te parece que el siguiente procedimiento responde a todas las especificaciones que hemos apuntado?

```
SEA MUEVELOS
HAZ "ACEL1 ACEL.MOV
HAZ "ACEL2 OPUESTO :ACEL1 ;[puedes probar tambien MULTIPLICA
0.7 OPUESTO ACEL1]
SINLAPIZ
COLOR :COLOR1
POS :POS1
CONLAPIZ
MUEVE1 :ACEL1
SINLAPIZ
COLOR :COLOR2
POS :POS2
CONLAPIZ
MUEVE2 :ACEL2
HAZ "POS1 :POS1.ACT
HAZ "VEL1 :VEL1.ACT
HAZ "POS2 :POS2.ACT
HAZ "VEL2 :VEL2.ACT
MUEVELOS
FIN
```







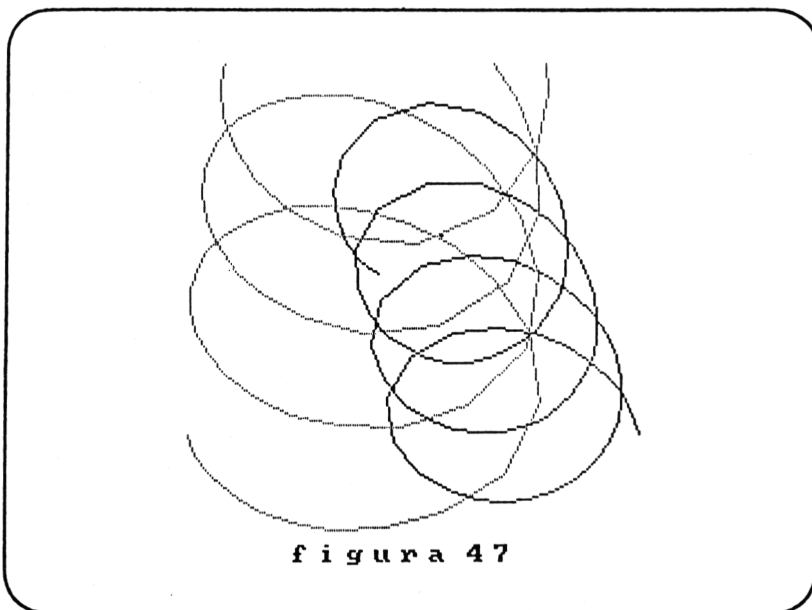


figura 47

Los movimientos no tienen problemas, simplemente

```
SEA MUEVE1 :ACEL
HAZ "VEL1.ACT SUMAR :VEL1 :ACEL
HAZ "POS1.ACT SUMAR :POS1 :VEL1.ACT
POS :POS1.ACT
FIN
```

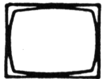
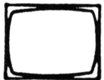
Ya sabes lo que tiene que hacer SUMAR. Escríbelo, a no ser que ya lo tengas. Termina de escribir y pon en marcha a los cuerpos celestes.

¿Te parece que se podría acortar el programa?

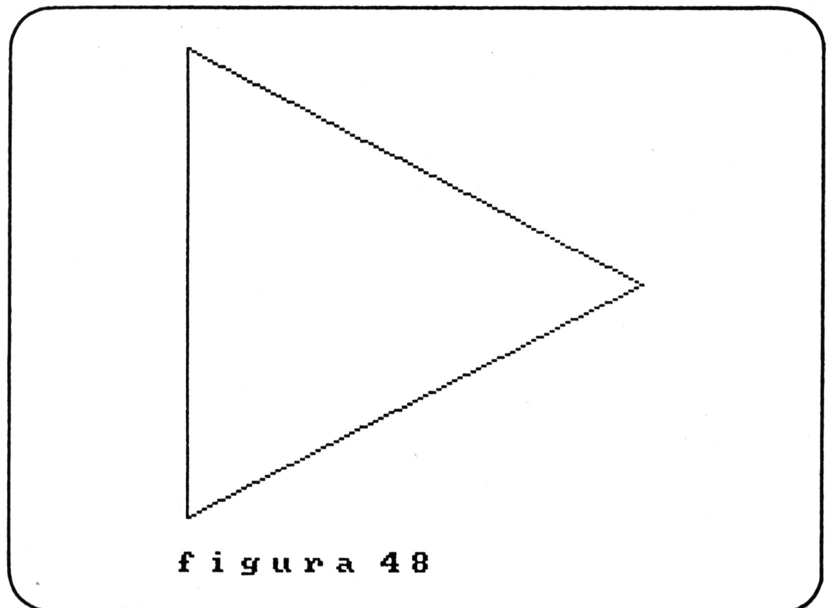
Podrías ver qué sistemas son estables y estudiar las relaciones entre distancias y aceleraciones para una ATRACCION dada. ¿Tiene algún significado físico la constante ATRACCION?

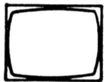
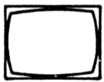
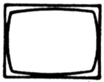
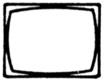
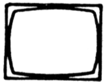
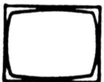
4.14 Curva sorprendente

Considera un triángulo equilátero. Sobre cada uno de sus lados construye otro, suprimiendo después el lado común, como se indica en la serie de figuras:





figura 48



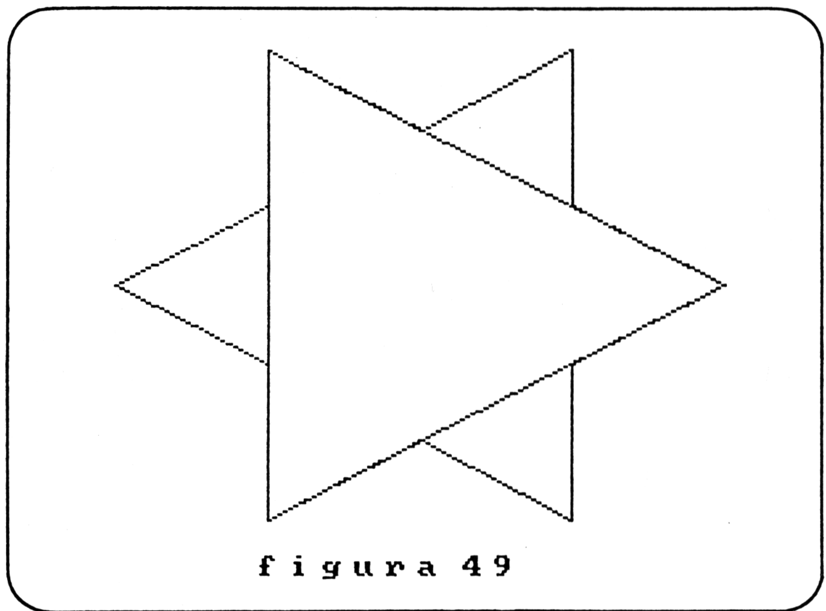



figura 49

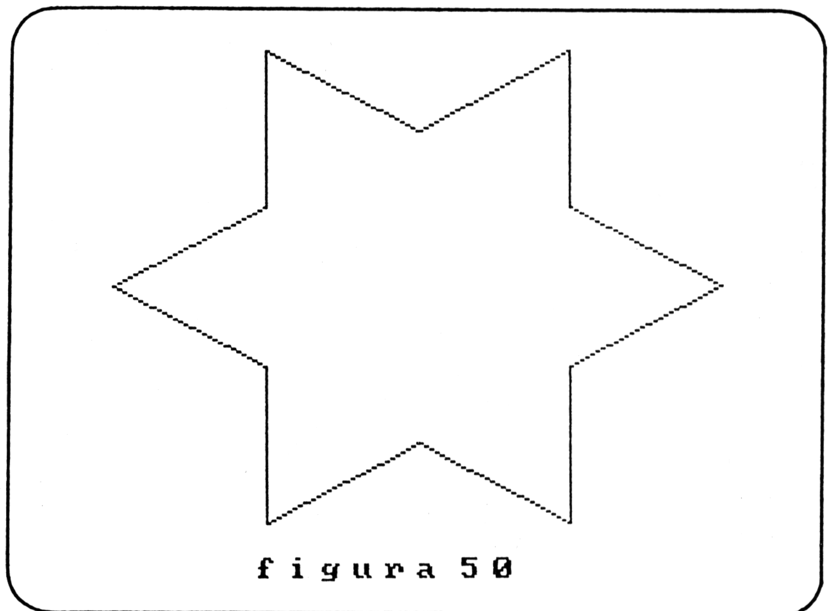


figura 50

Vuelve a hacer lo mismo con la última figura, y así sucesivamente. Imagínate la curva que se obtiene al continuar indefinidamente este proceso.

Veamos un programa que dibuja curvas de este tipo.

El triángulo equilátero es

```
SEA C1 :LONG
REPITE 3 [AVANZA :LONG DERECHA 120]
FIN
```

La segunda figura sería, utilizando C1,

```
SEA C2 :LONG
REPITE 3 [LADO DERECHA 120]
FIN
```

Puesto que el procedimiento LADO se tendría que encargar de dibujar los nuevos lados, podrías poner, fijándote en un lado de la segunda figura,

```
SEA LADO :LONG
AVANZA :LONG / 3
IZQUIERDA 60
AVANZA :LONG / 3
DERECHA 120
AVANZA :LONG / 3
IZQUIERDA 60
AVANZA :LONG / 3
FIN
```

Comprueba ahora que C2 produce la figura que tienes como ejemplo. Para la C3 va a ser más cómodo utilizar un procedimiento recursivo, que ya se anunciaba en la frase "y así sucesivamente". Utilizando un NIVEL como ya hicimos al dibujar árboles, si hacemos que el nivel 0 sea el triángulo equilátero, tendremos:

```
SEA CS :LONG :NIVEL
REPITE 3 [LADO :LONG :NIVEL DERECHA 120]
FIN

SEA LADO :LONG :NIVEL
SI :NIVEL = 0 [AVANZA :LONG ALTO]
LADO :LONG / 3 :NIVEL - 1
IZQUIERDA 60
LADO :LONG / 3 :NIVEL - 1
DERECHA 120
LADO :LONG / 3 :NIVEL - 1
IZQUIERDA 60
LADO :LONG / 3 :NIVEL - 1
FIN
```

Comprueba que con CS 200 0 obtienes el triángulo equilátero, y con CS 200 1 obtienes la última figura del ejemplo. Prueba algunos niveles consecutivos más. Si lo haces observarás que, al aumentar el valor de NIVEL, aumenta el área encerrada por la curva, así como su perímetro.

¿Qué relación hay entre el área de un nivel y la del siguiente? ¿Y entre los perímetros? Si encuentras estas relaciones, cosa no muy difícil

teniendo en cuenta cómo se pasa de un nivel al siguiente, podrías predecir lo que ocurrirá al aumentar más y más el nivel.

Si te resulta difícil encontrar las relaciones, siempre te queda el recurso de "la fuerza bruta". Modifica el programa para que te dé el perímetro (es sencillo: sólo tienes que sumar todos los AVANZA) y el área (algo más complicado, ya que tienes que añadir el área de un triángulo equilátero de lado :LONG, que es 0.25 * $\sqrt{3}$ * :LONG * :LONG cada vez que añadas un nuevo triángulo).

Insistimos. Al aumentar el valor de NIVEL, ¿crees que el área aumentará más y más, o bien tendrá un límite? ¿Y el perímetro?

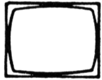
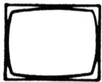
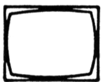
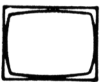
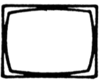
Lo sorprendente de esta curva y de otras análogas, en las que a lo mejor ya estás pensando, estriba precisamente en el "contraste" entre las respuestas a estas dos preguntas. No te vamos a decir cuáles son para no privarte del placer de la sorpresa.

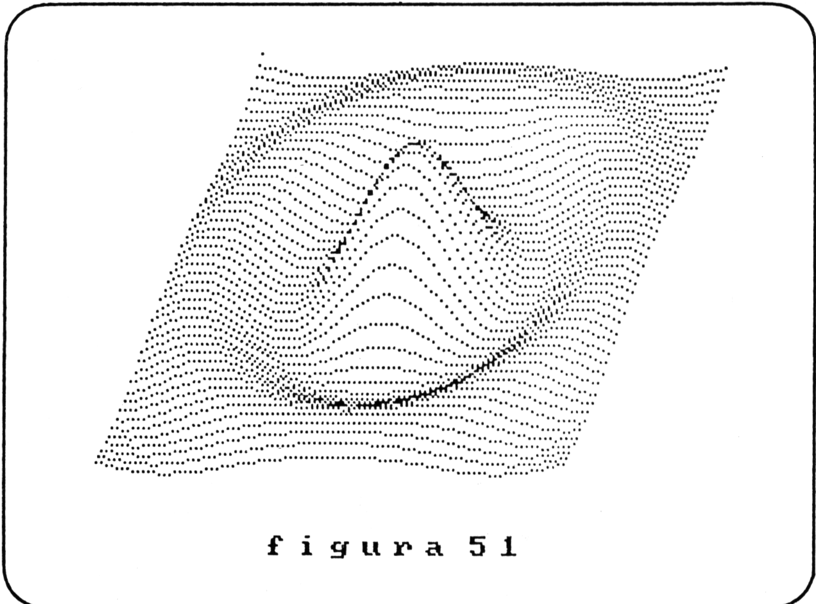
4.15

Tres dimensiones

Si te gusta la gráfica adjunta, que corresponde a la función tridimensional

```
Z = (SENO R) / R  donde R = √(X² + Y²)
```



te diremos cómo hacerte un programa que dibuja funciones de este estilo, que pueden escribirse en la forma Z = F(X,Y).

Una aclaración antes de comenzar: las funciones SEN y COS de LOGO tienen su entrada en grados, mientras que las utilizadas habitualmente en matemáticas la tienen en radianes. Debes construir los procedimientos SENO y COSENO cuya entrada sea en radianes y transformar éstos en grados para pasárselos a la correspondiente función de LOGO. Recuerda que π radianes equivalen a 180 grados. De manera que haremos el procedimiento SENO

```
SEA SENO :VALOR
DEVUELVE SEN 57.296 * :VALOR
FIN
```

y de modo análogo el del COSENO.

Podrías empezar TRESD de manera similar a los programas de gráficas: pidiendo la expresión de la función y traduciéndola a la notación LOGO. No olvides que ahora tendrás que cambiar no sólo X, sino también Y y R, si es necesario.

Para el ejemplo anterior te sugerimos esta respuesta a la pregunta sobre la expresión de la función

```
HAZ "R RAC (X * X + Y * Y) HAZ "Z (SENO R) / R
```

con lo que se calcula R una sola vez.







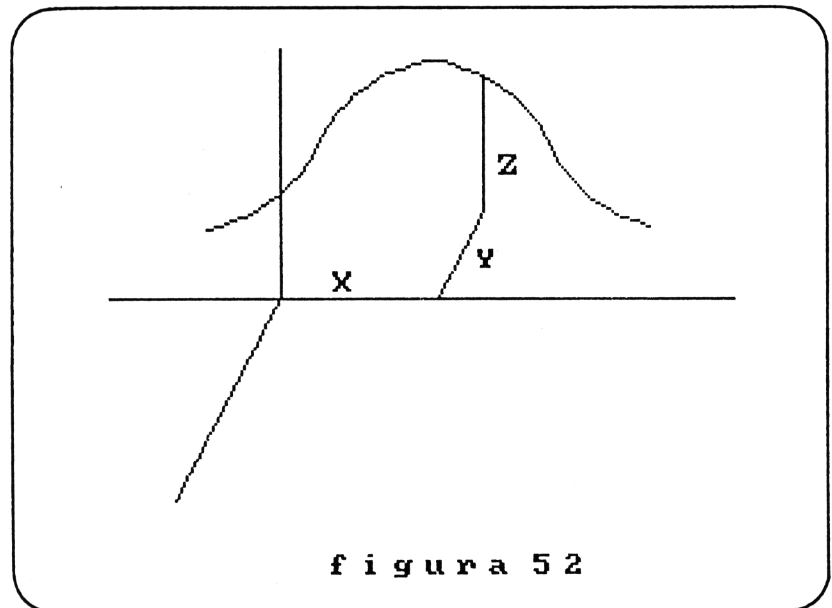


figura 52

Puesto que ya disponemos de la expresión de Z, puedes empezar a dar valores. Para cada par X Y obtendrás un valor de Z como se muestra en la figura 52, en la que C es una curva de la superficie obtenida haciendo variar la X para un valor de Y fijo. Si se hace variar la Y se obtienen una serie de curvas que dan la sensación de volumen. El truco consiste en representar, para cada par (X,Y), el punto (X,Y+Z). Para conseguir una sensación de perspectiva de los ejes puede aplicarse lo que técnicamente se conoce con el nombre de "cizalladura de ángulo α". De modo que el punto de coordenadas (X1,Y1) se transforma en el coordenadas (X2,Y2), cumpliéndose las relaciones siguientes:

```
X2 = X1 + Y1 * COS α
Y2 = Y1 * SEN α
```

como puedes ver en la figura

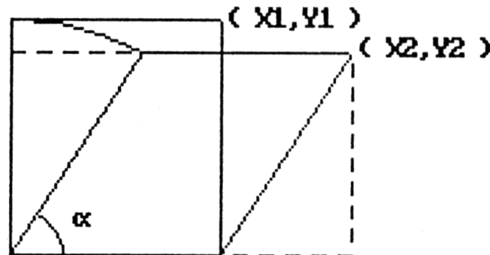


figura 53

Teniendo esto presente, dado el punto (X,Y), calculamos Z y representamos en la pantalla el punto (XP,YP) tal que

```
XP = X + (Y * COSALFA)
YP = (Y * SENALFA) + Z
```

siendo SENALFA y COSALFA, SEN α y COS α, respectivamente.
El programa TRESD podría tener entonces el procedimiento LEEFUNCION y el EMPIEZA. Este último haría lo siguiente:

Pedir la escala para los ejes X e Y, almacenándola.
Lo mismo, para la función, el eje Z.
Pedir el ángulo para la perspectiva y calcular SENALFA y COSALFA.
Limpiar la pantalla gráfica.
Llamar a DIBUJA, que se encargará de dibujar la superficie.
Ofrecer opciones de impresión, almacenamiento en disco y continuación con otras escalas.

Dejando el resto a tu iniciativa, un posible DIBUJA sería:

```
SEA DIBUJA :X1 :Y1
(LOCAL "X "Y)
SI :Y1 > 90 [ALTO]
SI :X1 > 90 [HAZ "X1 -90 HAZ "Y1 :Y1 + 4]
HAZ "X :X1 * :ESCALAXY
HAZ "Y :Y1 * :ESCALAXY
COGE "ERROR [PINTA]
DIBUJA :X1 + 2 :Y1
FIN

SEA PINTA
(LOCAL "Z1 "XP "YP)
EJECUTA :FUNCION
HAZ "Z1 :Z * ESCALAZ
HAZ "XP :X1 + (:Y1 * :COSALFA)
HAZ "YP (:Y1 * :SENALFA) + :Z1
PUNTO LISTA :XP :YP
FIN
```

Si pones en marcha el programa TRESD comprobarás que el resultado es diferente del correspondiente a la superficie del ejemplo. La diferencia se debe a que se ven puntos que no deberían verse, ya que están "tapados por otros más altos". Esto es porque el programa no entiende de "altos y bajos". Hay una solución para evitarlo: cuando se va a dibujar el punto (XP,YP) se comprueba si, para ese valor de XP, se ha dibujado ya algún Y mayor que YP; si no se ha hecho, se dibuja el punto y se almacena el valor de YP para posteriores comparaciones, y si se ha hecho, no se dibuja el punto. La forma de llevarlo a la práctica está en almacenar los valores de YP sucesivos en una variable que sirva de control de altura, CA, y escribir:

```
SI VALOR :CA < :YP [PUNTO LISTA :XP :YP HAZ :CA :YP]   (1)
```

Es evidente que, para empezar, hay que asignar valores a todos los posibles CA. Si echas cuentas comprobarás que el total de los que necesitas es

```
REDONDEA 90 * (1 + COSALFA)
```

de modo que puedes usar la variable TOTAL, local a EMPIEZA, que almacene este número, y llamar en él al procedimiento CONTRO-

LALT, con los valores 0 y :TOTAL como entrada. Este último procedimiento podría ser:

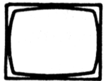
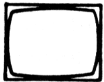
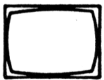
```
SEA CONTROLALT :C :TOT
SI :C > :TOT [ALTO]
HAZ "C -90
CONTROLALT :C + 1 :ALT
FIN
```

Antes de utilizar la línea (1) debes escribir

```
HAZ "CA :TOTAL + :REDONDEA :XP
```

Si incorporas estas nuevas instrucciones obtendrás un programa para dibujo de superficies bastante decente. Su principal problema es la lentitud. Y no podemos remediarlo por el momento. Puedes ponerlo en marcha e irte a comer. Otro problema en el APPLE es que, al no disponer de funciones exponenciales, hay que programarlas; entonces sí que puede eternizarse.

Puesto que en IBM puedes almacenar en disco dibujos de pantalla, podrías hacerlo con los de la misma función usando distintas escalas, y pasar después todos los dibujos seguidos, consiguiendo así un efecto como si te alejases de la superficie. Hablando de escalas, el dibujo de la introducción está hecho usando



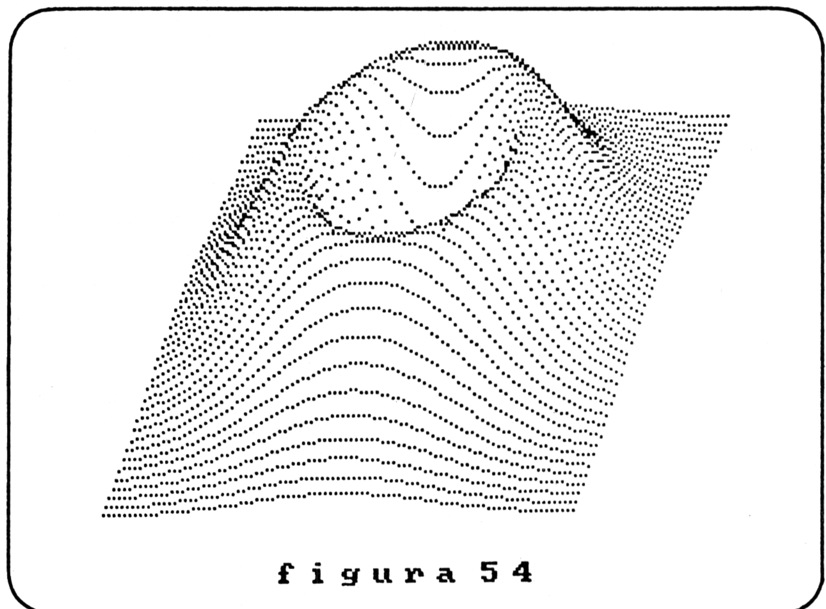


figura 54

- ESCALAXY 0.1
- ESCALAZ 50
- ANGULO para la perspectiva 70

Si dispones de un IBM utiliza los valores 0.12, 0.06, 0.04, 0.02, 0.01, para ESCALAXY, junto con 50 y 70 para ESCALAZ y ANGULO, respectivamente, en la función

$$Z = (R * (R - 2) * EXP\ -0.5 * R)^2, \text{ donde } R = \sqrt{(X^2 + Y^2)}$$

Obtendrás figuras como la 54.

4.16

Resolución de ecuaciones

Utilizando los programas GRAFICAS puedes tener una idea de cuáles son las soluciones de la ecuación $\sqrt{x} = x - 1$. Basta representar la función $y = \sqrt{x} - x + 1$, ya que las soluciones de la ecuación son los puntos de intersección de la función con el eje X.

Puesto que en un dibujo no se obtienen los valores con mucha precisión, vamos a describir una técnica sencilla que permite determinar una solución, con la precisión que quieras, si sabes en qué intervalo está, y esto sí que te lo dice el dibujo.

La técnica se basa en una propiedad de las funciones continuas definidas en un intervalo, que dice lo siguiente:

Si una función toma valores de distintos signos en los extremos de un intervalo, entonces existe un punto interior en el que la función toma el valor 0. Este punto interior es la solución.

Según esto, si sabes que la función tiene "un 0" en el intervalo [a,b], se determina el punto medio $c = (a+b)/2$. Si en c tiene distinto signo que en a, la solución está en [a,c] y, en caso contrario, en [c,b]. Observa que así se ha conseguido reducir el intervalo a la mitad. De modo que repitiendo la misma operación con el nuevo intervalo reducimos éste otra vez a la mitad, obteniendo uno que es ya la cuarta parte del original. Si continuamos así iremos reduciendo cada vez más la longitud del intervalo en el que se halla la solución, hasta que la diferencia entre ambos extremos sea más pequeña que un cierto número. El punto medio de ese intervalo nos da una aproximación de la solución con la precisión deseada. Si, por ejemplo, hemos fijado la precisión en 0.001 y llegamos al intervalo [2.1436 2.1438] podemos dar como solución 2.1437 con la mencionada precisión, de modo que 2.144 es el redondeo a milésimas.

Necesitamos, pues, un procedimiento que realice las particiones sucesivas del intervalo :IZQ :DER. ¿Te sirve éste?

```
SEA PARTICION :IZQ :DER
HAZ "X (IZQ + :DER) / 2 SI DIFERENCIA :DER :IZQ < :PRECISION
[ESCRIBE FRASE [LA SOLUCION APROXIMADA ES] :X ALTO]
HAZ "Y EJECUTA :F
SI IGUALP :SIGNO (:Y < 0) [HAZ "IZQ :X] [HAZ "DER :X]
PARTICION :IZQ :DER
FIN
```

SIGNO es una variable cuyos valores serán VERDAD o FALSO, que debe tener ya un valor. Este valor puede dársele en el procedimiento que lance a partición; por ejemplo,

```
SEA RESUELVE
(LOCAL "X "Y "SIGNO)
HAZ "X :EXTREMO.IZQ
HAZ "Y EJECUTA :F
HAZ "SIGNO :Y < 0
PARTICION :EXTREMO.IZQ :EXTREMO.DER
FIN
```

y, en definitiva, SOLUCION podría ser:

```
SEA SOLUCION
LEEFUNCION
EMPIEZA
TERMINA
FIN
```

LEEFUNCION seguramente lo conoces, pues se ha utilizado bastante. EMPIEZA debe pedir los extremos del intervalo y la precisión, lanzando entonces a RESUELVE. TERMINA se encarga de preguntar si se ha finalizado, se cambia de función o de intervalo, etcétera.

Antes de que empieces con ellos es oportuno hacer una observación sobre PARTICION. ¿Qué ocurre si Y vale 0, es decir, si el valor de X calculado es exactamente la solución? El procedimiento continuaría partiendo el intervalo sin enterarse. Convendrá, pues, hacer un test sobre Y = 0; si es verdad, hay que detenerse y dar la solución, y si es falso, se continúa como antes. No te decimos que lo efectúes también en RESUELVE, ya que es poco probable que si un número aparece como posible solución lo utilices como extremo del intervalo. Incorpora enseguida esta nueva condición.

Si no quieres usar el programa GRAFICAS para tener una idea de por dónde anda la solución, podrías modificarlo para que te permitiera buscar en distintos intervalos, indicándote en cuáles no puede asegurarse que exista solución.

Observa que, tal y como está el programa, si le das un intervalo en el que no haya una solución, el programa sigue “rodando” indefinidamente. Puedes modificar RESUELVE en la forma siguiente:

Se ve el signo en IZQ tal como se hace en el procedimiento, almacenándolo en SIGNOI, y después en DER, almacenándolo en SIGNOD. Si ambos coinciden no es seguro que haya solución y se

debe escribir el mensaje oportuno, comenzando de nuevo. Si son distintos se continúa como antes. Puesto que el arreglo es fácil, incorpóralo ahora mismo, no sin revisar los demás procedimientos por si hay que cambiar algo.

Elige ecuaciones más "extrañas" y ponlo en marcha.

4.17 Traza

Es posible que en algunas ocasiones un programa no funcione exactamente como tú querías que lo hiciera, sin que sepas en qué momento se ha apartado del "recto camino".

A veces basta con observar los valores que se asignan a las variables para salir del atolladero, pero HAZ no dice una palabra de lo que hace. Se le puede redefinir para que lo haga. Por ejemplo, si HAZ fuese el procedimiento

```
SEA HAZ :NOMBRE :VALOR
ESCRIBE []
(ESCRIBE :NOMBRE [TOMA EL VALOR] :VALOR)
ESCRIBE []
.HAZ :NOMBRE :VALOR
FIN
```

en el que .HAZ realiza el trabajo que antes efectuaba HAZ, podrías ver qué valores van asignándose a las variables. Para poder definir este procedimiento necesitas:

- Poder redefinir una primitiva, lo que se consigue escribiendo

REDEFP

```
HAZ "REDEFP "VERDAD
```

- Copiar HAZ en .HAZ, para lo que basta poner

COPIADEF

```
COPIADEF ".HAZ "HAZ
```

- Dar la nueva definición de HAZ, es decir,

```
DEFINE "HAZ [[NOMBRE  VALOR]  [ESCRIBE []] [(ESCRIBE :NOMBRE
[TOMA EL VALOR] :VALOR)] [ESCRIBE []] [.HAZ :NOMBRE :VALOR]]
```

Si después de haber hecho todo esto pones en marcha tu programa, siempre que en él aparezca HAZ funcionará el procedimiento que acabas de definir, y podrás comprobar qué valores toman las variables. Una vez hayas terminado, debes dejar las cosas como estaban, para lo cual basta con escribir:

```
COPIADEF "HAZ ".HAZ
BORRA ".HAZ
HAZ "REDEFP "FALSO
```

Si esto no es suficiente, lo que tú necesitas es una traza del programa. Una traza de un programa te informa de:

> El procedimiento que va a trazar, diciéndote qué valores toman las variables de la cabecera.
> La línea que va a ejecutar en cada momento.

De este modo es posible seguir paso a paso la evolución del programa y localizar las desviaciones con respecto a la idea del programador.

Para trazar un determinado programa hay que fabricar, a partir de él, uno que se comporte de la manera indicada.

Por ejemplo, para trazar POLI construiremos el programa .POLI que haga la traza de sí mismo. La ejecución de .POLI nos indicará cómo funciona POLI y, una vez hayamos terminado con él, bastará con eliminarlo.

Puesto que un programa suele tener varios procedimientos, construiremos para cada uno el procedimiento .⟨nombre⟩; estos nombres precedidos de un punto los iremos almacenando en .LISTAREDEF para su eliminación posterior. El programa TRAZA, al que incorporaremos la utilidad .HAZ y un COGE "ERROR para evitar problemas, podría empezar así:

```
SEA TRAZA
(LOCAL ".LINEA ".MENSAJE ".LISTAREDEF)
HAZ ".LISTAREDEF "FALSO
ESCRIBE [ESCRIBE EL PROCEDIMIENTO A TRAZAR COMO SI FUERAS A
EJECUTARLO]
HAZ ".LINEA LEELISTA
SI MIEMBROP :.LINEA [[TRAZA] [NO TRACES]] [ALTO]
HAZ "REDEFP "VERDAD
COPIADEF ".HAZ "HAZ
DEFINE "HAZ <ya sabes como>
COGE "ERROR [.TRAZALINEA :.LINEA]
COPIADEF "HAZ ".HAZ
BORRA ".HAZ
HAZ "REDEFP "FALSO
HAZ ".MENSAJE ERROR
SI NO VACIAP :.MENSAJE [ESCRIBE [] ESCRIBE SINPRIMERO
:.MENSAJE ESCRIBE []]
SI NO VACIAP :.LISTAREDEF [BORRA :.LISTARREDEF]
ESCRIBE []
TRAZA
```

Como hay que preparar el programa "trazable" —lo que lleva algún tiempo— y ejecutarlo, vamos a poner

```
SEA .TRAZALINEA :LINEA
ESCRIBE FRASE [ESPERA UN MOMENTO MIENTRAS PREPARO LA TRAZA
DE] :LINEA
EJECUTA .PROCESALINEA :LINEA
FIN
```

.PROCESALINEA debe devolver la línea procesada lista para su ejecución. Utilizando la variable local .PROCESADA podríamos poner:

```
SEA .PROCESALINEA :LINEA
LOCAL ".PROCESADA
.HAZ ".PROCESADA []
.PROCESA :LINEA
DEVUELVE :.PROCESADA
FIN
```

.PROCESA debe procesar uno a uno los elementos de la línea, hasta terminar con ella, colocando los resultados en .PROCESADA

```
SEA .PROCESA :LINEA
LOCAL ".PRIMERO
SI VACIAP :LINEA [ALTO]
.HAZ ".PRIMERO PRIMERO :LINEA
TEST LISTAP :.PRIMERO
SIVERDAD [.HAZ ".PROCESADA PONULTIMO (.PROCESA :PRIMERO)
:.PROCESADA]
SIVERDAD [.PROCESA SINPRIMERO :LINEA]
SIFALSO [.HAZ ".PRIMERO .PROCESAPALABRA :.PRIMERO]
SIFALSO [.HAZ ".PROCESADA PONULTIMO :.PRIMERO :.PROCESADA]
.PROCESA SINPRIMERO :LINEA
FIN
```

.PROCESAPALABRA debe encargarse de reconocer si una palabra es el nombre de uno de los procedimientos del programa, y en tal caso hacer lo siguiente:

> Definir el procedimiento "trazable" correspondiente, cuyo nombre será .⟨nombre⟩.
> Almacenar .⟨nombre⟩ en .LISTAREDEF.
> Devolver .⟨nombre⟩.

Cualquier palabra que pase el test DEFINIDOP ⟨palabra⟩ es un nombre de procedimiento, pero antes de trazarlo hay que comprobar si está en .LISTAREDEF; si es así, es que ya ha sido trazado y basta con devolver ⟨palabra⟩, mientras que si no está hay que hacer las tres operaciones que hemos indicado antes. Ten en cuenta que HAZ es ahora un procedimiento y no debes trazarlo, ya que tiene su propia traza.

Las palabras que no pasen el test es porque son primitivas (recuerda que .HAZ es ahora una primitiva), en cuyo caso basta con devolver la palabra en cuestión. Podríamos escribir:

```
SEA .PROCESAPALABRA :PAL
LOCAL ".PROCEDIMIENTO
SI (O (NO DEFINIDOP :PAL) (PRIMITIVAP :PAL) (:PAL = "HAZ))
[DEVUELVE :PAL]
.HAZ ".PROCEDIMIENTO PALABRA ". :PAL
SI (Y (DEFINIDOP :PAL) (MIEMBROP :.PROCEDIMIENTO
:.LISTAREDEF)) [DEVUELVE :.PROCEDIMIENTO]
```

```
.HAZ ".LISTAREDEF PONULTIMO :.PROCEDIMIENTO :.LISTAREDEF
DEFINE :.PROCEDIMIENTO .TRAZAPROCE :PAL
DEVUELVE :.PROCEDIMIENTO
FIN
```

Vamos a construir, pues, la traza del procedimiento. Si al preguntar el programa que queremos trazar hemos contestado con

```
ESPI 10 89 4
```

pretendemos que cuando empiece a trabajar produzca algo como

```
EJECUTO ESPI
LADO VALE 10
ANGULO VALE 89
INCRE VALE 4
AVANZA :LADO
```

aquí esperará a que se pulse una tecla y, cuando se haga, ejecutará la orden, es decir, la tortuga hará su pequeño recorrido.

```
DERECHA :ANGULO
```

otra vez esperará

```
EJECUTO ESPI
LADO VALE 14
ANGULO VALE 89
INCRE VALE 4
```

etcétera.

Hay que distinguir entonces entre la cabecera del procedimiento, donde están las variables, y el cuerpo, donde están las instrucciones.

TEXPRO

La primitiva TEXPRO proporciona el texto de un procedimiento como una lista de listas en la que la primera de ellas contiene los nombres de los parámetros, y las demás, las instrucciones del procedimiento. A partir de este texto construiremos uno nuevo, insertando las cosas que deseamos, que será lo que devuelva .TRAZAPROCE. Así pues,

```
SEA .TRAZAPROCE :PROCE
(LOCAL ".TEXTO ".NUEVOTEXTO ".VARIABLES)
.HAZ ".TEXTO TEXPRO :PROCE
.HAZ ".NUEVOTEXTO []
.HAZ ".VARIABLES PRIMERO :.TEXTO
.HAZ ".NUEVOTEXTO PONULTIMO :.VARIABLES :.NUEVOTEXTO
```

En este momento .NUEVOTEXTO es [[LADO ANGULO INCRE]], que son los parámetros del nuevo procedimiento, y debemos incluir en él, como una lista, la instrucción que produce en la pantalla:

```
EJECUTO ESPI
```

que sería

```
ESCRIBE FRASE "EJECUTO :PROCE
```

de modo que continuaríamos así:

```
HAZ ".NUEVOTEXTO PONULTIMO  (LISTA "ESCRIBE "FRASE ""EJECUTO
:.NUEVOTEXTO)
.TRAZACABECERA :.VARIABLES
.TRAZACUERPO SINPRIMERO :.TEXTO
DEVUELVE :.NUEVOTEXTO
FIN
```

No estaría mal incluir, antes de esta última parte, un

```
.HAZ ".NUEVOTEXTO PONULTIMO (LISTA "ESCRIBE []) :.NUEVOTEXTO
```

¿Comprendes por qué?

.TRAZACABECERA tiene que ir escribiendo el valor de cada variable, como has visto en el ejemplo; así pues,

```
SEA .TRAZACABECERA :VAR
SI VACIAP :VAR [ALTO]
.HAZ  ".NUEVOTEXTO PONULTIMO (LISTA "( "ESCRIBE PRIMERO :VAR
""VALE PALABRA ": PRIMERO :VAR ") ) :.NUEVOTEXTO
.TRAZACABECERA SINPRIMERO :VAR
FIN
```

Vamos con el cuerpo del procedimiento, en el que hay que incluir:

La escritura de la instrucción.
El procedimiento que detenga la ejecución hasta que se pulse una tecla.
La propia instrucción a ejecutar.

```
SEA .TRAZACUERPO :CUERPO
SI VACIAP :CUERPO [ALTO]
.HAZ ".NUEVOTEXTO PONULTIMO (LISTA "ESCRIBE PRIMERO :CUERPO)
:.NUEVOTEXTO
.HAZ ".NUEVOTEXTO PONULTIMO [.ESPERATE] :.NUEVOTEXTO
.HAZ ".NUEVOTEXTO PONULTIMO PRIMERO :CUERPO :.NUEVOTEXTO
.TRAZACUERPO SINPRIMERO :CUERPO
FIN
```

.ESPERATE debe limitarse a esperar a que se pulse una tecla; queda a tu cargo.

Increíble pero cierto: ¡ya está!

Para utilizarlo es mejor que introduzcas todo en el paquete TRAZA, escribiendo

PQTODO

```
PQTODO "TRAZA
```

EPS

Si escribes EPS (abreviatura para Escribe PropiedadeS), te encontrarás con un montón de líneas del tipo

```
.TRAZALINEA --> PROCPQ ES TRAZA
```

lo que te indica que .TRAZALINEA es un procedimiento del paquete TRAZA, y lo mismo para los demás procedimientos. Una vez empaquetado se procede a ocultarlo, escribiendo:

OCULTA

```
OCULTA "TRAZA
```

Si ahora escribes ECTS descubrirás que ¡no hay ninguno! Normal. ¿No los has ocultado? Pero algo queda de todo ello; escribe EPS y te encontrarás con

```
TRAZA --> OCULTA ES VERDAD
```

Para guardarlo en el disco, debes escribir:

```
GUARDA <nombre del archivo> <nombre del paquete>
```

En este caso te aconsejamos que escribas el mismo nombre en ambos, para no tener problemas a la hora de traerlo; esto es:

```
GUARDA "TRAZA "TRAZA
```

Más adelante, cuando quieras trazar un programa te bastará con hacer un simple TRAE "TRAZA y lo tendrás listo para su uso.

Tras esta digresión sobre el almacenamiento de TRAZA, volvamos con él. Si lo has probado habrás observado dos cosas, al menos, que requieren un retoque.

Una es el tiempo que hay que esperar mientras se construye el programa trazable. La solución, mejor que escribir un mensaje que solicite paciencia del usuario, es que se vayan escribiendo los nombres de los procedimientos que se van trazando, lo cual nos lleva a la otra. No funciona "tal y como decía la propaganda", aunque a esto ya debes estar acostumbrado. El que ejecute, sin trazarlos, procedimientos que están dentro de otros se debe a que, en .TRAZACUERPO, la línea

```
.HAZ ".NUEVOTEXTO PONULTIMO PRIMERO :CUERPO :.NUEVOTEXTO
```

coloca, sin más, una línea del antiguo procedimiento en el nuevo; pero esa línea puede ser ESPI :LADO :ANGULO :INCRE, con lo cual en el nuevo procedimiento no aparecerá el mensaje EJECUTO ESPI con los valores que tienen en ese momento las variables. Para que esto

ocurra habría que procesar PRIMERO :CUERPO como lo que es: una línea. ¿Crees que las cosas se arreglarían escribiendo en lugar de la línea anterior esta otra?

```
.HAZ  ".NUEVOTEXTO  PONULTIMO  .PROCESALINEA PRIMERO :CUERPO
:.NUEVOTEXTO
```

Por probar no pierdes nada.

Si usas TRAZA es porque tienes que hacer correcciones en tu programa, de modo que podrías incorporar algunas de las primitivas de manejo de procedimientos y archivos, tal que, sin salir de TRAZA, pudieras escribir cosas como EDITA "ESPI para pasar al editor. Quizá bastaría con un

```
SI MIEMBROP PRIMERO :.LINEA [EDITA ECTS  ECVS  ECTODO  BORRA
BOVS BOTODO BOARCHIVO GUARDA TRAE] [EJECUTA :.LINEA ALTO]
```

en el lugar adecuado de TRAZA.

El tema ha sido largo, pero no todos los días se elabora un programa que modifique a otro.

4.18
Experto

Sólo los seres inteligentes razonan. Esta frase y alguna de sus variantes la habrás escuchado montones de veces. Si un programa es capaz de resolver problemas de lógica, al menos como un alumno de BUP, ¿lo considerarás inteligente? ¿Por qué? Una discusión de este tipo puede durar lo que se quiera, pero vamos a centrarnos en lo que nos interesa. Se trata de construir un programa que "razone". (No entramos en el problema de la construcción de tablas de verdad de proposiciones, que está "tirado".)

Vamos a precisar utilizando un sencillo ejemplo de lógica:

Supongamos que dispones de los siguientes hechos, escritos en notación prefija

IMPLICA A B
IMPLICA B C
NO C

¿Qué se deduce de aquí? Nada, si no dispones de reglas que aplicar. Vamos a dar una:

SI IMPLICA X Y e IMPLICA Y Z entonces IMPLICA X Z.

Usando esta regla se obtendría un nuevo hecho, IMPLICA A C, con lo cual ya disponemos de una base más amplia:

IMPLICA A B
IMPLICA B C
NO C
IMPLICA A C

¿Y ahora qué? Añadamos una nueva regla:

SI IMPLICA X Y y NO Y entonces NO X.

Aplicada a la nueva base produciría dos nuevos hechos: NO B, NO A. Tendríamos, en definitiva, la base

IMPLICA A B
IMPLICA B C
NO C
IMPLICA A C
NO B
NO A

Para empezar, ¿cómo vamos a darle los hechos al programa? Escribiremos el hecho en notación prefija, o pulsaremos simplemente el "retorno de carro", sin escribir nada, para indicar que hemos terminado. Cuando reciba éste, escribirá la base que le hemos proporcionado. Con objeto de ampliar la base cuando queramos, podemos escribir:

```
SEA CRAE.BASE
SI NO VALORP "BASEO [HAZ "BASEO []]
CREA.BASE.CONT
FIN
```

Recuerda que VALORP es la primitiva que devuelve "VERDAD si su entrada tiene asignado un valor, y "FALSO en caso contrario.

```
SEA CREA.BASE.CONT
LOCAL "HECHO
(TECLEA [¿HECHO?] CAR 32)
HAZ "HECHO LEELISTA
TEST :HECHO = []
SIVERDAD [ESCRI.BASE :BASEO]
SIFALSO [HAZ "BASEO PONULTIMO :HECHO :BASEO]
SIFALSO [CREA.BASE.CONT]
FIN
```

ESCRI.BASE no presenta problemas, ya que se trata simplemente de escribir los elementos de una lista. Queda a tu cargo.

La cuestión de las reglas es un poco más delicada. Para facilitar la

tarea del usuario, una vez que haya decidido si escribe o no una nueva regla, conviene separar el antecedente del consecuente, incluso éstos entre sí, terminando la introducción de antecedentes y consecuentes con un "retorno de carro" (↵). Por ejemplo,

```
¿ESCRIBES? (S / N)  (pulsas S)
¿ANTECEDENTE? IMPLICA #X #Y
¿ANTECEDENTE? IMPLICA #Y #Z
¿ANTECEDENTE? (pulsas retorno)
¿CONSECUENTE? IMPLICA #X #Z
¿CONSECUENTE? (pulsas retorno)
¿ESCRIBES? (S / N)  (pulsa S o N, según desées)
```

Se pretende que el antecedente quede almacenado como

[[IMPLICA #X #Y][IMPLICA #Y #Z]]

y el consecuente como

[[IMPLICA #X #Z]]

La regla estará almacenada como [antecedente consecuente] dentro de la lista REGLAS.

Antes de comentar el signo #, vamos a escribir los procedimientos correspondientes:

```
SEA CREA.REGLAS
SI NO VALORP "REGLAS [HAZ "REGLAS []]
CREA.REGLAS.CONT
FIN

SEA CREA.REGLAS.CONT
(LOCAL "ANTECEDENTE "CONSECUENTE)
HAZ "ANTECEDENTE []
HAZ "CONSECUENTE []
ESCRIBE [¿ESCRIBES? (S/N)]
TEST LEECAR = "N
SIVERDAD [ESCRI.REGLAS :REGLAS]
SIFALSO [LEE.ANTE]
FIN

SEA LEE.ANTE
LOCAL "ANTE
(TECLEA [¿ANTECEDENTE?] CAR 32)
HAZ "ANTE LEELISTA
TEST :ANTE = []
SIFALSO [HAZ "ANTECEDENTE PONULTIMO :ANTE :ANTECEDENTE
LEE.ANTE]
SIVERDAD [LEE.CONSE]
FIN
```

LEE.CONSE es análogo, pero si el test sobre :CONSE = [ ] es verdad, esto es, se ha terminado de escribir la regla, la instrucción correspondiente sería

```
HAZ "REGLAS PONULTIMO (LISTA :ANTECEDENTE :CONSECUENTE)
:REGLAS CREA.REGLAS.CONT
```

ESCRI.REGLAS es análogo a ESCRI.BASE. Conviene, debido a la estructura de las reglas, introducirle el procedimiento ESCRI.REGLA :REGLA, cuya única instrucción sería

```
ESCRIBE (FRASE PRIMERO :REGLA [=>] ULTIMO :REGLA)
```

Vamos con el signo #. Cualquier letra o palabra precedida por este signo indicará que se trata de una variable. La esencia del programa, como verás más adelante, es comparar las reglas con los hechos fijándose si coinciden los prefijos. En caso afirmativo asocia a las variables los valores que figuran en los hechos. Pero vamos por partes. Puesto que tenemos los hechos y las reglas, sólo falta aplicarlas, así que

```
SEA APLICA :REGLAS
TEST VACIAP :REGLAS
SIVERDAD [ESCRIBE [] ESCRIBE [NUEVA BASE:] ESCRI.BASE :BASEO
ESCRIBE []]
SIFALSO [PREPARA PRIMERO :REGLAS]
SIFALSO [APLICA SINPRIMERO :REGLAS]
FIN
```

Como ves, se trata de aplicar sucesivamente cada una de las reglas a la base y, cuando se haya terminado, escribir la nueva base.

PREPARA realizará lo siguiente:

- Descomponer cada regla en antecedente y consecuente.
- Deducir, en la forma ya expuesta, un resultado del antecedente y cada uno de los hechos de la base.
- Generar un nuevo hecho utilizando este resultado.

Así pues,

```
SEA PREPARA :REGLA
(LOCAL "ANTECEDENTE "CONSECUENTE "RESULTADO)
HAZ "ANTECEDENTE PRIMERO :REGLA
HAZ "CONSECUENTE ULTIMO :REGLA
HAZ "RESULTADO []
(TECLEA [APLICO:] CAR 32)
ESCRI.REGLA :REGLA
DEDUCE :ANTECEDENTE :BASE []
GENERA :RESULTADO
FIN
```

DEDUCE va a construir una lista del tipo

[[#X A][#Y B][#Z C]],

lista de las variables con sus valores correspondientes, utilizando la técnica de la "unificación" *(matching)*, que constituye el quid de la cuestión.

La idea de la unificación es la siguiente:

Se compara un predicado del antecedente, IMPLICA #X #Y, con un hecho de la base, IMPLICA A B; puesto que ambos coinciden en su primer elemento, proseguimos con la unificación construyendo la pareja variable-valor [#X A]; es decir, la variable X toma el valor A, y se continúa así hasta lograr la unificación del predicado y el hecho. Así pues, DEDUCE va a trabajar hasta que termine con el antecedente, construyendo un resultado formado por el consecuente y la lista variables-valores, realizando esta tarea con todos los hechos de la base hasta que termine con ella:

```
SEA DEDUCE :ANTE :BASE :LISTA.VARVAL
LOCAL "LISTA.PROV
SI :BASE = [] [ALTO]
TEST :ANTE = []
SIVERDAD [HAZ "RESULTADO PONPRIMERO (LISTA :CONSECUENTE
:LISTA.VARVAL) :RESULTADO]
SIFALSO [DEDUCE1]
FIN
```

DEDUCE1 va a intentar la unificación del primero de la lista antecedente con el primero de la lista de hechos; si la unificación falla pasará al hecho siguiente, manteniendo el mismo antecedente. Si la unificación tiene éxito aplicamos DEDUCE, con el resto del antecedente, a toda la base. Esto es necesario, ya que de lo contrario fallarían las unificaciones de hechos del tipo

NO C
IMPLICA A C

con la segunda regla que dimos antes al estar las premisas en orden inverso al que están en la regla. Después se aplica DEDUCE con el antecedente al resto de la base, sin tener en cuenta los resultados de la unificación previa por si pueden conseguirse nuevas unificaciones. En resumen:

```
SEA DEDUCE1
HAZ "LISTA.PROV :LISTA.VARVAL
TEST UNIFICACION PRIMERO :ANTE :PRIMERO :BASE
SIVERDAD [DEDUCE SINPRIMERO :ANTE :BASE :LISTA.VARVAL]
SIVERDAD [DEDUCE :ANTE SINPRIMERO :BASE :LISTA.PROV]
SIFALSO [DEDUCE :ANTE  SINPRIMERO :BASE :LISTA.VARVAL]
FIN
```

UNIFICACION tiene que realizar dos cometidos: construir la lista variables-valores y devolver VERDAD o FALSO. Sus dos entradas son un predicado del antecedente, [IMPLICA #X #Y], y un hecho de la base, [IMPLICA A B]. Si el primer elemento del predicado coincide con el primero del hecho, devolverá el resultado de aplicar UNIFICACION sin los primeros de ambos, con objeto de continuar la misma. Si no coinciden, #X y A, hay que verificar si lo que está

manejando del predicado es una variable; ése es el papel del símbolo #. Identificar las variables es inmediato usando

```
SEA ESVARIABLE :OBJ
DEVUELVE (PRIMERO :OBJ = "#)
FIN
```

Pues bien, si no se trata de una variable hay que devolver FALSO, ya que son distintos. Si la variable se puede unificar con el valor, se unifica y se continúa con el resto del predicado, devolviéndose FALSO en caso contrario. Puesto que de esta manera o se devuelve FALSO en algún momento, o se termina con el predicado, cuando esto ocurra se devuelve VERDAD.

```
SEA UNIFICACION :PREDICADO :HECHO
SI :PREDICADO = [] [DEVUELVE "VERDAD]
SI PRIMERO :PREDICADO = PRIMERO :HECHO [DEVUELVE UNIFICACION
SINPRIMERO :PREDICADO SINPRIMERO :HECHO]
TEST ESVARIABLE PRIMERO :PREDICADO
SIVERDAD [SI UNIFICA PRIMERO :PREDICADO :PRIMERO :HECHO
[DEVUELVE UNIFICACION SINPRIMERO :PREDICADO SINPRIMERO
:HECHO] [DEVUELVE "FALSO]]
SIFALSO [DEVUELVE "FALSO]
FIN
```

UNIFICA tiene dos entradas, una variable y un valor, debiendo devolver VERDAD si puede asignarse ese valor a la variable, y FALSO en caso contrario. Si la variable no ha recibido todavía ningún valor, sí se puede unificar, añadiéndose el par variable-valor a la lista variables-valores. Si, por el contrario, ya tiene un valor asociado, habrá que devolver VERDAD si coincide con el de entrada, y FALSO en caso contrario

```
SEA UNIFICA :VAR :VAL
LOCAL "VALOR.VAR
HAZ "VALOR.VAR ASOCIA :VAR :LISTA.VARVAL
TEST :VALOR.VAR = "
SIVERDAD [HAZ "LISTA.VARVAL PONPRIMERO (LISTA :VAR :VAL)
:LISTA.VARVAL]
SIVERDAD [DEVUELVE "VERDAD]
SIFALSO [DEVUELVE :VAL = :VALOR.VAL]
FIN
```

ASOCIA debe buscar en la lista variables-valor el valor correspondiente a la variable de entrada, para devolverlo si lo encuentra o, en caso contrario, devolver la palabra vacía:

```
SEA ASOCIA :VAR :LISTA.VV
SI :LISTA.VV = [] [DEVUELVE " ]
TEST :VAR = PRIMERO PRIMERO :LISTA.VV
SIVERDAD [DEVUELVE ULTIMO PRIMERO :LISTA.VV]
SIFALSO [DEVUELVE ASOCIA :VAR SINPRIMERO :LISTA.VV]
FIN
```

El largo camino de la deducción ha terminado.

A estas alturas RESULTADO es una lista formada por el consecuente y la lista variables-valor; por ejemplo,

[[[IMPLICA #X #Z]] [[#X A] [#Y B] [#Z C]]]

con la que debe generarse el hecho

IMPLICA A C

para incorporarlo a la base.

Vamos por partes; procesamos la primera lista de RESULTADO y continuamos con el resto.

```
SEA GENERA :RESULTADO
SI :RESULTADO = [] [ALTO]
GENERA1 (PRIMERO PRIMERO :RESULTADO) (ULTIMO PRIMERO
:RESULTADO)
GENERA SINPRIMERO :RESULTADO
FIN
```

GENERA1 se va a encargar de comparar cada predicado de la lista de consecuentes con la lista variables-valor para obtener hechos:

```
SEA GENERA1 :CONSECUENTES :LISTA.VARVAL
TEST :CONSECUENTES = []
SIFALSO [GENERA2 PRIMERO :CONSECUENTES LISTA.VARVAL []]
SIFALSO [GENERA1 SINPRIMERO :CONSECUENTES :LISTA.VARVAL]
FIN
```

GENERA2 tiene que encargarse de construir el hecho a partir del predicado examinándolo "palabra por palabra". Si la palabra es una variable, debe colocar su valor. Pero si no lo es (por ejemplo, IMPLICA), debe colocarla en HECHO. Cuando haya terminado debe poner HECHO en la base:

```
SEA GENERA2 :PREDICADO :LISTA.VV :HECHO
TEST :PREDICADO = []
SIVERDAD [SI NO (HECHO = []) [PONHECHO :HECHO]
SIFALSO [SI ESVARIABLE PRIMERO :PREDICADO [HAZ "HECHO
PONULTIMO (ASOCIA PRIMERO :PREDICADO :LISTA.VV) :HECHO] [HAZ
"HECHO PONULTIMO PRIMERO :PREDICADO :HECHO]]
SIFALSO [GENERA2 SINPRIMERO :PREDICADO :LISTA.VV :HECHO]
FIN
```

PONHECHO va a colocar el hecho en la base, si no lo está ya, y escribirlo al mismo tiempo en la pantalla para que se vea cómo progresa el programa:

```
SEA PONHECHO :HECHO
TEST NO (MIEMBROP :HECHO :BASEO)
SIVERDAD [ESCRIBE [] (ESCRIBE [NUEVO HECHO:] :HECHO) ESCRIBE
[]]
SIVERDAD [HAZ "BASEO PONULTIMO :HECHO :BASEO]
FIN
```

Ya está terminado. Sólo falta que funcione. Por cierto, ¿qué va a pasar en el ejemplo de lógica que pusimos al principio, si se escriben las reglas en orden inverso al que tenían en él? Cuando apliques la regla

[IMPLICA #X #Y] [NO #Y] ⇒ [NO #X]

a la base

IMPLICA A B
IMPLICA B C
NO C

sólo va a obtener el nuevo hecho

NO B

que incorporará a la base.

Si a continuación se aplica la regla

[IMPLICA #X #Y] [IMPLICA #Y #Z] ⇒ [IMPLICA #X #Z]

a esta nueva base, se obtendrá el nuevo hecho IMPLICA A C y se escribirá la nueva base de datos, ya que se han terminado las reglas. Esta nueva base es

IMPLICA A B
IMPLICA B C
NO C
NO B
IMPLICA A C

¿Dónde está el hecho NO A que aparecía antes en las deducciones? En ningún sitio, no aparece en este caso, al menos hasta que no vuelvas a escribir

```
APLICA :REGLAS
```

Entonces, la aplicación de la primera regla actual produce el esperado NO A, ya que ahora figuran en la base NO C e IMPLICA A C. La aplicación de la segunda no produce ningún hecho nuevo. Si vuelves a escribir APLICA :REGLAS no obtendrás ya nada nuevo. Se han terminado todas las posibles deducciones con esas reglas. Es fácil automatizar este proceso, guardando BASE0 en BASE1, usando

APLICA y comparando la nueva BASE0 con BASE1; si no son iguales, se vuelve a utilizar APLICA, y si lo son, se han terminado las deducciones:

```
SEA REAPLICA
LOCAL "BASE1
HAZ "BASE1 :BASE0
APLICA :REGLAS
TEST :BASE1 = BASE0
SIVERDAD [ESCRIBE [] ESCRIBE [NO PUEDO DEDUCIR NADA NUEVO]]
SIFALSO [REAPLICA]
FIN
```

Aunque hemos desarrollado el programa utilizando prácticamente las notaciones lógicas clásicas, puedes comprobar que, además de admitir como predicados y variables notaciones más acordes con el lenguaje natural, también se pueden manipular pequeñas bases de datos definiendo como reglas relaciones entre los objetos de la base.

Comprueba la siguiente base:

```
ENTRADA ENSALADA
ENTRADA ESPARRAGOS
PLATO TERNERA.EN.SALSA
PLATO ENTRECOT
POSTRE HELADO
POSTRE FLAN
```

con la regla

[ENTRADA #X] [PLATO #Y] [POSTRE #Z] ⇒ [COMIDA #X #Y #Z]

y verás el montón de menús que obtienes.

Introduce esta otra base:

```
PADRE PEDRO JUAN
PADRE JUAN LUIS
PADRE LUIS LUCAS
PADRE LUCAS JOSE
```

con estas reglas

[PADRE #X #Y] ⇒ [HIJO #Y #X]
[PADRE #X #Y] [PADRE #Y #Z] ⇒
[ABUELO #X #Z]
[ABUELO #X #Y] ⇒ [NIETO #Y #X]

y sabrás quién es quién en la familia.

Si estás pensando que se han acabado los problemas de lógica, te equivocas. Pronto descubrirás que el Experto no es tan experto como parece.

Claro está que si quieres arreglarlo...

Apéndice
DICCIONARIO DE PRIMITIVAS

Castellano-Inglés

ABIERTOS		ALLOPEN		*
ABRE		OPEN		*
ALTO		STOP		
ANCHURA		SETWIDTH		*
ANULPROP		REMPROP		
ARCHIVOP		FILEP		
ARCTAN		ARCTAN		
ARRANCA		STARTUP		
ASCII		ASCII		
AVANZA	AV	FORWARD	FD	
AZAR		RANDOM		
BARRERA		FENCE		
BOARCHIVO		ERASEFILE		*
BOLAPIZ	BL	PENERASE	PE	
BOPS		ERPS		
BORRA	BO	ERASE	ER	
BOTODO		ERALL		
BOTONP		BUTTOMP		
BOV		ERN		
BOVS		ERNS		

* Solamente en I.B.M
\# Solamente en Apple

CAR		CHAR		
CENTRO		HOME		
CIERRA		CLOSE		*
CIERRATODO		CLOSEALL		*
CO		CO		
COCIENTE		QUOTIENT		
COGE		CATCH		
COLOR		SETPC		
COLORTEXTO	CT	SETTC		*
CONLAPIZ	CL	PENDOWN	PD	
COPIADEF		COPYDEF		
COS		COS		
CUENTA		COUNT		
CURSOR		SETCURSOR		
DEFINE		DEFINE		
DEFINIDOP		DEFINEDP		
DERECHA	DE	RIGHT	RT	
DESCUBRE		UNBURY		
DEVUELVE	DV	OUTPUT	OP	
DIFERENCIA		DIFFERENCE		*
DIR		DIR		
DISCO		SETDISK		
ECARCHIVO		POFILE		*
ECP		PO		
ECPS		POPS		
ECTODO		POALL		
ECTS		POTS		
ECVS		PONS		
EDARCHIVO		EDITFILE		*
EDITA	ED	EDIT	ED	
EDVS		EDNS		
EJECUTA		RUN		
ENT		INT		
ENVIA		THROW		
EPS		PPS		
ERRACT		ERRACT		
ERROR		ERROR		
ESCALA		SETSCRUNCH		#
ESCRIBE	EC	PRINT	PR	
ESCRIBIR		SETWRITE		*
ESCRITURA		WRITER		*
ESPERA		WAIT		
ESPIA		DRIBBLE		*
ET		HIDETURTLE	HT	
ETIQUETA		LABEL		
EXP		EXP		*

FALSO		FALSE		
FIN		END		
FINECP		WRITEOFP		*
FINLECP		READEOFP		*
FONDO		SETBG		
FORMA		SETSHAPE		*
FORMATO		FORM		*
FRASE	FR	SENTENCE	SE	
GRAFICOS	GR	FULLSCREEN	FS	
GUARDA		SAVE		
GUARDADIB		SAVEPIC		*
HACIA		TOWARDS		
HAZ		MAKE		
HUELLA		STAMP		*
IGUALP		EQUALP		
INAZAR		RERANDOM		
INVLAPIZ	IL	PENREVERSE	PX	
ITEM		ITEM		
IZQUIERDA	IZ	LEFT	LT	
LAPIZ		SETPEN		
LECTURA		READER		*
LEECAR	LC	READCHAR	RC	
LEECARS	LCS	READCHARS	RCS	*
LEELISTA	LL	READLIST	RL	
LEEPALABRA	LP	READWORD	RW	*
LEER		SETREAD		*
LG		CLEARSCREEN	CS	
LIBERA		RECYCLE		
LIMPIA		CLEAN		
LINTEXTO		SETTEXT		*
LISTA		LIST		
LISTAP		LISTP		
LLAMA		NAME		
LN		LN		*
LOCAL		LOCAL		
LONGARCHIVO		FILELEN		*
LPROP		PLIST		
LTEXTO	LT	CLEARTEXT	CT	
MANDOP		PADDLE		
MAYUSC		SETCAPS		*
MAYUSCP		CAPS		*
MIEMBROP		MEMBERP		
MIXTA	MX	MIXEDSCREEN	MS	

MODELO		SNAP		*
MT		SHOWTURTLE	ST	
MUESTRA		SHOW		
NIVELSUP		TOPLEVEL		
NO		NOT		
NODOS		NODES		
NOESPIA		NODRIBBLE		*
NOTACION		EFORM		*
NUMEROP		NUMBERP		
O		OR		
OCULTA		BURY		
PALABRA		WORD		
PALABRAP		WORDP		
PALETA		SETPAL		*
PAQUETE		PACKAGE		
PAUSA		PAUSE		
PI		PI		*
PONPRIMERO		FPUT		
PONULTIMO		LPUT		
POS		SETPOS		
POSEC		SETWRITEPOS		*
POSLEC		SETREADPOS		*
POSX		SETX		
POSY		SETY		
POTENCIA		POWER		*
PPROP		PPROP		
PQTODO		PKGALL		
PRECISION		SETPRECISION		*
PRIMERO		FIRST		
PRIMITIVAP		PRIMITIVEP		
PROCPQ		PROCPKG		
PRODUCTO		PRODUCT		
PUNTO		DOT		
RAC		SQRT		
RECOMPILA		REPARSE		
REDEFP		REDEFP		
REDONDEA		ROUND		
RELLENA		FILL		*
REPITE		REPEAT		
RESTO		REMAINDER		
RETROCEDE	RE	BACK	BK	
RUMBO		SETHEADING	SETH	
SEA		TO		

SEN		SIN		
SI		IF		
SIFALSO	SIF	IFFALSE	IFF	
SINLAPIZ	SL	PENUP	PU	
SINPRIMERO	SP	BUTFIRST	BF	
SINULTIMO	SU	BUTLAST	BL	
SIVERDAD	SIV	IFTRUE	IFT	
SUMA		SUM		
TA		TURTLE		*
TECLAP		KEYP		
TECLEA		TYPE		
TEST		TEST		
TEXPRO		TEXT		
TEXTOS	TX	TEXTSCREEN	TS	
TONO		TONE		*
TORTUGAP		SHOWNP		
TRAE		LOAD		
TRAEDIB		LOADPIC		*
ULTIMO		LAST		
VACIAP		EMPTYP		
VALANCHURA		WIDTH		*
VALAPIZ		PEN		
VALCOLOR	VC	PENCOLOR	PC	
VALCT		TEXTCOLOR	TC	*
VALCURSOR		CURSOR		
VALDISCO		DISK		
VALESCALA		SCRUNCH		#
VALFONDO	VF	BACKGROUND	BG	*
VALFORMA		SHAPE		*
VALOR		THING		
VALORP		NAMEP		
VALPALETA		PALETTE		*
VALPOS		POS		
VALPOSEC		WRITEPOS		*
VALPOSLEC		READPOS		*
VALPRECISION		PRECISION		*
VALPROP		GPROP		
VALRUMBO		HEADING		
VALX		XCOR		
VALY		YCOR		
VARPQ		VALPKG		
VE		GO		
VENTANA		WINDOW		
VERDAD		TRUE		
VUELVE		WRAP		

Y	AND	
.BGUARDA	.BSAVE	*
.BTRAE	.BLOAD	*
.COM	.SETCOM	*
.CONTENIDO	.CONTENTS	
.DOS	.DOS	*
.ESCALA	.SETSCRUNCH	*
.EXAMINA	.EXAMINE	
.PANT	.SETSCREEN	*
.PON	.DEPOSIT	
.PRINTER	.IMPRESORA	#
.PTA	.BPT	#
.SISTEMA	.SYSTEM	
.VALESCALA	.SCRUNCH	*
.VALPANT	.SCREEN	*
.VE	.CALL	*

Inglés-Castellano

ALLOPEN		ABIERTOS		*
AND		Y		
ARCTAN		ARCTAN		
ASCII		ASCII		
BACK	BK	RETROCEDE	RE	
BACKGROUND	BG	VALFONDO	VF	
BURY		OCULTA		
BUTFIRST	BF	SINPRIMERO	SP	
BUTLAST	BL	SINULTIMO	SU	
BUTTOMP		BOTONP		
CAPS		MAYUSCP		*
CATCH		COGE		
CHAR		CAR		
CLEAN		LIMPIA		
CLEARSCREEN	CS	LG		
CLEARTEXT	CT	LTEXTO	LT	
CLOSE		CIERRA		*
CLOSEALL		CIERRATODO		*
CO		CO		

* Solamente en I.B.M.
Solamente en Apple.

COPYDEF		COPIADEF		
COS		COS		
COUNT		CUENTA		
CURSOR		VALCURSOR		
DEFINE		DEFINE		
DEFINEDP		DEFINIDOP		
DIFFERENCE		DIFERENCIA		*
DIR		DIR		
DISK		VALDISCO		
DOT		PUNTO		
DRIBBLE		ESPIA		*
EDIT	ED	EDITA	ED	
EDITFILE		EDARCHIVO		*
EDNS		EDVS		
EFORM		NOTACION		*
EMPTYP		VACIAP		
END		FIN		
EQUALP		IGUALP		
ERALL		BOTODO		
ERASE	ER	BORRA	BO	
ERASEFILE		BOARCHIVO		*
ERN		BOV		
ERNS		BOVS		
ERPS		BOPS		
ERRACT		ERRACT		
ERROR		ERROR		
EXP		EXP		*
FALSE		FALSO		
FENCE		BARRERA		
FILELEN		LONGARCHIVO		*
FILEP		ARCHIVOP		
FILL		RELLENA		
FIRST		PRIMERO		
FORM		FORMATO		*
FORWARD	FD	AVANZA	AV	
FPUT		PONPRIMERO		
FULLSCREEN	FS	GRAFICOS	GR	
GO		VE		
GPROP		VALPROP		
HEADING		VALRUMBO		
HIDETURTLE	HT	ET		
HOME		CENTRO		

IF		SI		
IFFALSE	IFF	SIFALSO	SIF	
IFTRUE	IFT	SIVERDAD	SIV	
INT		ENT		
ITEM		ITEM		
KEYP		TECLAP		
LABEL		ETOQUETA		
LAST		ULTIMO		
LEFT	LT	IZQUIERDA	IZ	
LIST		LISTA		
LISTP		LISTAP		
LN		LN		*
LOAD		TRAE		
LOADPIC		TRAEDIB		*
LOCAL		LOCAL		
LPUT		PONULTIMO		
MAKE		HAZ		
MEMBERP		MIEMBROP		
MIXEDSCREEN	MS	MIXTA	MX	
NAME		LLAMA		
NAMEP		VALORP		
NODES		NODOS		
NODRIBBLE		NOESPIA		*
NOT		NO		
NUMBERP		NUMEROP		
OPEN		ABRE		*
OR		O		
OUTPUT	OP	DEVUELVE	DV	
PACKAGE		PAQUETE		
PADDLE		MANDO		
PALETTE		VALPALETA		*
PAUSE		PAUSA		
PEN		VALAPIZ		
PENCOLOR	PC	VALCOLOR	VC	
PENDOWN	PD	CONLAPIZ	CL	
PENERASE	PE	BOLAPIZ	BL	
PENREVERSE	PX	INVLAPIZ	IL	
PENUP	PU	SINLAPIZ	SL	
PI		PI		*
PKGALL		PQTODO		
PLIST		LPROP		
PO		ECP		

POALL		ECTODO		
POFILE		ECARCHIVO		*
PONS		ECVS		
POPS		ECPS		
POS		VALPOS		
POTS		ECTS		
POWER		POTENCIA		*
PPROP		PPROP		
PPS		EPS		
PRECISION		VALPRECISION		*
PRIMITIVEP		PRIMITIVAP		
PRINT	PR	ESCRIBE	EC	
PROCPKG		PROCPQ		
PRODUCT		PRODUCTO		
QUOTIENT		COCIENTE		
RANDOM		AZAR		
READCHAR	RC	LEECAR	LC	
READCHARS	RCS	LEECARS	LCS	*
READEOFP		FINLECP		*
READER		LECTURA		*
READLIST	RL	LEELISTA	LL	
READPOS		VALPOSLEC		*
READWORD	RW	LEEPALABRA	LP	*
RECYCLE		LIBERA		
REDEFP		REDEFP		
REMAINDER		RESTO		
REMPROP		ANULPROP		
REPARSE		RECOMPILA		
REPEAT		REPITE		
RERANDOM		INAZAR		
RIGHT	RT	DERECHA	DE	
ROUND		REDONDEA		
RUN		EJECUTA		
SAVE		GUARDA		
SAVEPIC		GUARDADIB		*
SCRUNCH		VALESCALA		#
SENTENCE	SE	FRASE	FR	
SETBG		FONDO		
SETCAPS		MAYUSC		*
SETCURSOR		CURSOR		
SETDISK		DISCO		
SETHEADING	SETH	RUMBO		
SETPAL		PALETA		*
SETPC		COLOR		
SETPEN		LAPIZ		

SETPOS		POS		
SETPRECISION		PRECISION		*
SETREAD		LEER		*
SETREADPOS		POSLEC		*
SETSCRUNCH		ESCALA		#
SETSHAPE		FORMA		*
SETTC		COLORTEXTO	CT	*
SETTEXT		LINTEXTO		*
SETWIDTH		ANCHURA		*
SETWRITE		ESCRIBIR		*
SETWRITEPOS		POSEC		*
SETX		POSX		
SETY		POSY		
SHAPE		VALFORMA		*
SHOW		MUESTRA		
SHOWNP		TORTUGAP		
SHOWTURTLE	ST	MT		
SIN		SEN		
SNAP		MODELO		*
SQRT		RAC		
STAMP		HUELLA		*
STARTUP		ARRANCA		
STOP		ALTO		
SUM		SUMA		
TEST		TEST		
TEXT		TEXPRO		
TEXTCOLOR	TC	VALCT		*
TEXTSCREEN	TS	TEXTOS	TX	
THING		VALOR		
THROW		ENVIA		
TO		SEA		
TONE		TONO		*
TOPLEVEL		NIVELSUP		
TOWARDS		HACIA		
TRUE		VERDAD		
TURTLE		TA		*
TYPE		TECLEA		
UNBURY		DESCUBRE		
VALPKG		VARPQ		
WAIT		ESPERA		
WIDTH		VALANCHURA		*
WINDOW		VENTANA		
WORD		PALABRA		
WORDP		PALABRAP		

WRAP	VUELVE	
WRITEOFF	FINECP	*
WRITEPOS	VALPOSEC	*
WRITER	ESCRITURA	*
XCOR	VALX	
YCOR	VALY	
.BLOAD	.BTRAE	*
.BPT	.PTA	#
.BSAVE	.BGUARDA	*
.CALL	.VE	*
.CONTENTS	.CONTENIDO	
.DEPOSIT	.PON	
.DOS	.DOS	*
.EXAMINE	.EXAMINA	
.PRINTER	.IMPRESORA	#
.SCREEN	.VALPANT	*
.SCRUNCH	.VALESCALA	*
.SETCOM	.COM	*
.SETSCREEN	.PANT	*
.SETSCRUNCH	.ESCALA	*
.SYSTEM	.SISTEMA	

Más que un lenguaje de programación, LOGO es una concepción de la enseñanza, basada en la utilización del ordenador como herramienta de trabajo y amplificador de la inteligencia, que nos permite explorar, investigar, resolver problemas; es decir, aprender.

Escrito para que puedas adentrarte sin dificultad en el uso del LOGO, PROGRAMACION EN LOGO te llevará de la mano desde tus primeros procedimientos hasta la construcción de un sistema experto escrito en LOGO.

El libro está dividido en 4 partes:

— Primeros pasos: donde se toma contacto con el mundo de la tortuga, el entorno interactivo y las primitivas básicas del lenguaje LOGO.

— Programando con procedimientos: que explica los fundamentos de la construcción de procedimientos y las estructuras fundamentales de programación.

— Números y listas: que se adentra en la manipulación de números y palabras.

— Miscelánea: excitante colección de programas LOGO cuyo desarrollo te proporcionará un considerable dominio del lenguaje y un primer contacto con problemas interesantes como: tortuga dinámica sin rozamiento, representaciones gráficas, traducción, inteligencia artificial (ELIZA), árboles, programación estructurada, resolución de ecuaciones, tres dimensiones, sistemas expertos.

PROGRAMACION EN LOGO utiliza una versión castellana de LOGO implementada en los ordenadores APPLE II e IBM PC por el Grupo GOLEM de Valencia.

ANAYA
MULTIMEDIA



Programación en LOGO

Joaquín D'Opazo Alvarez/Grupo Golem